LinuxFP: Transparently Accelerating Linux Networking

Marcelo Abranches
University of Colorado, Boulder
Marcelo.DeAbranches@colorado.edu

Erika Hunhoff
University of Colorado, Boulder
Erika.Hunhoff@colorado.edu

Rohan Eswara
University of Colorado, Boulder
Rohan.Eswara@colorado.edu

Oliver Michel
Princeton University
omichel@princeton.edu

Eric Keller
University of Colorado, Boulder
Eric.Keller@colorado.edu

***Abstract*—This paper introduces transparent acceleration into the Linux networking stack. To do so, we build on years of research in creating high-performance software-based packet processing systems. Rather than treating these technologies as alternative pipelines, we leverage the technology to create explicit fast paths in the Linux kernel. With this, Linux still serves as a complete implementation of all its supported protocols, but frequent operations on the critical path can be transparently handled by a fast path. We implement a controller that continuously introspects the Linux kernel to determine exactly what packet-processing functionality is currently configured. The controller then synthesizes and deploys a minimal fast past into the packet processing pipeline that only implements functionality that is currently needed. In this way, common command line tools, such as brctl, control plane software, such as FRRouting (FRR), and higher-level management frameworks such as Kubernetes and Ansible, work without modification and transparently benefit from a faster network data plane. Our system, LinuxFP, includes a controller that can implement IP forwarding, bridging, and IP filtering fast paths that are synthesized on-demand using their specific and current configuration in the kernel. We evaluate performance improvements using Linux management tools and a Kubernetes network plugin. We show performance improvements over Linux for packet forwarding of 77% and 20% for an unmodified Kubernetes network plugin.**

I. Introduction

Software-based packet processing is widely adopted across a number of use cases, such as data-center load balancing [24], virtualized networking between containers [43] or virtual machines [33], multi-cloud overlay networking [5], and 5G infrastructures [26]. Software-based packet processing requires both high performance and, in many cases, the ability to introduce custom functionality. While Linux is the most widely used platform for many such use cases, supporting the required packet processing performance with its out-of-the-box networking stack is challenging [30]. Likewise, introducing custom functionality is generally considered painful, requiring, for example, kernel recompilation or additional kernel modules that hook into potentially evolving internal interfaces and structures.

This led to the development of frameworks that enable developers to write high-performance software-based packet processing applications. These take several different forms, such as (1) kernel bypass (e.g., DPDK [21] or netmap [42]), where packets are efficiently copied into user space for processing, (2) in-kernel network stack bypass (e.g., XDP [30] or Click [32]), which present alternative in-kernel pipelines for packet processing which typically ensure packets do not touch the Linux networking stack, or (3) as a new kernel (e.g., x-kernel [31] or Demikernel [45]). These frameworks provide the required API and blueprint for high-performance packet-processing applications, but developers are left to implement both the complete packet-processing pipeline and control-plane integration.

To lower the barrier to leveraging these high-performance packet-processing frameworks, new platforms were introduced to provide common functionality on top of these framewoks. Polycube [37] is a kernel-space platform built around eBPF technology that includes coarse-grained network functions which can be configured using custom command line tools. VPP [9] is a user-space platform leveraging kernel bypass technology (e.g., DPDK [21]) that also provides custom management tools for configuring network functions. Each system operates as a separate packet processing pipeline from the Linux networking stack, and use of the system requires use of custom APIs and tooling.

Lack of compatibility with the Linux networking API is both a problem and a lost opportunity. There is a rich ecosystem of widely-used and extensively tested software that includes management tools (e.g., iproute2 [27]), control-plane software (e.g., FRR [28]), and container orchestration platforms (e.g., Kubernetes [12]). These tools are all dependent on Linux networking APIs. Ideally, a system would be able to support this ecosystem while also supporting acceleration.

To this end, we introduce LinuxFP, a system which enables fast packet processing while retaining the sophisticated networking capabilities of Linux. The key approach of LinuxFP is to transparently[1] accelerate hot spots in the Linux packet-processing stack using an explicit fast path to accelerate common-case processing. This is a widely-used design pattern

[1]Transparent, in this case, refers to view point of the user, who will not see any difference in how they interact with Linux.

in computer networking, where the common case is heavily optimized while a slow path handles less frequent corner cases and complex tasks, such as processing control-plane messages or handling packet fragmentation. There were two key challenges of realizing this design in LinuxFP: 1) *how* to accelerate, and 2) *what* to accelerate.

The question of *how* to accelerate is deeper than just picking a high-performance packet-processing technology, such as DPDK or eBPF. To accelerate packet processing while supporting all existing Linux networking functionality and the Linux networking API, LinuxFP uses Linux as the default slow path for packet processing and selectively installs eBPF-based fast paths as needed to accelerate common functionality. The code that defines the fast path accesses kernel network state and configuration (e.g., routing entries or firewall rules) instead of using custom data structures and a custom control plane. Coherent state across the slow and fast paths is critical for ensuring correctness when packets may be processed on different paths.

To determine *what* to accelerate, we design a controller that continuously inspects state in the Linux kernel to determine what packet-processing functionality is currently configured and used. LinuxFP then synthesizes and deploys only fast path components that would be used by the current configuration, which keeps the fast path configuration light-weight, meaning a minimal critical path in the data plane.

We built a complete working prototype of LinuxFP, supporting accelerating within Linux kernel 6.6 for bridging, IP forwarding, and IP filtering. LinuxFP is evaluated on two real scenarios (a virtual router and a virtual gateway), where LinuxFP is configured only using standard Linux techniques. In contrast to Linux, LinuxFP is 77% faster for forwarding with 53% lower latency. To determine whether there is a performance impact of the state sharing between the fast and slow paths, we compare LinuxFP against the alternative accelerated packet-processing platform Polycube. We show that LinuxFP does not see lower performance than Polycube[2]. While LinuxFP is configured using standard Linux configuration tools, Polycube was configured with its custom interfaces and management software. We also evaluate another common use of Linux networking, a Kubernetes network plugin. We show a speedup over Linux of 20% and latency reduction of 18% for pod-to-pod communication with an unmodified network plugin (Flannel).

In this paper, we first provide background on fast packet processing frameworks and platforms, motivating the need for LinuxFP (Section II). We then introduce the design goals and architecture of LinuxFP in Section III. Next, we provide a deeper discussion of the complete system design (Section IV) and implementation (Section V). We evaluate LinuxFP in different scenarios (Section VI) and discuss related work (Section VII). We conclude in Section VIII. Code for LinuxFP is available at https://github.com/mcabranches/tna.

[2]LinuxFP actually sees a throughput improvement of 19% over Polycube, but we attribute that to implementation differences.

II. Background / Motivation

Due to the importance of software-based packet processing, there is a history of strong research in this space. Here, we discuss previous works and identify a significant gap that this work seeks to fill.

A. Fast Software Packet Processing

We first look at advances in the performance of software-based packet processing technologies. While there is a great number of works in this area, we highlight three significant advances.

Click Modular Router. The Click modular router [32], or just Click, was introduced to address the problem of routers being closed and inflexible, whereas users needed flexibility and extensibility. Click is a framework allowing a developer or admin to specify a data flow graph of packet processing modules called *elements*. Developers *click* together elements to define the functionality of the router, and can extend the router by introducing new elements as C++ classes. While designed for flexibility, Click provided high-performance - achieving 4x performance improvements over Linux for similar functionality. The authors attributed the performance due to device handling improvements from previous works [22], [38]. We consider Click to be significant in the fast packet processing space because it provided a complete framework which allowed networking researchers to easily create new high-performance packet processing applications and led to many years of high-impact research.

DPDK. The Data Plane Development Kit (DPDK) [21] was introduced as a framework for user space packet processing which is both safer, due to the isolation of processes, and easier to debug. While Click was capable of processing packets in user space, the performance was substantially lower than doing so in kernel space. DPDK introduced a set of libraries that optimize user space networking through the introduction of kernel bypass technology and efficient data structures. Developers use the DPDK libraries to create custom pipelines.

eBPF/XDP. A challenge with DPDK is that it needed to bypass the kernel to obtain performance, but in doing so, sacrificed the ability to effectively interface with Linux. eBPF is a Linux technology for safely loading code into the kernel at various hook points. For networking, the eXpress Data Path (XDP) [30] introduced a hook point for creating optimized data paths for fast packet processing. Safety is provided through an in-kernel verifier of bytecode. Code can interface to the rest of the Linux kernel through (i) helper functions integrated into the kernel (e.g., for accessing the forwarding information base, or FIB), and (ii) packet interfaces to pass packets to the kernel. In a sense, XDP provides the best properties of both Click and DPDK: XDP is extensible, hooks into the kernel, and is safe and fast.

However, Click, DPDK, and XDP are enabling technologies, not platforms. While they enable creating packet-processing pipelines, extra work – sometimes significant – is needed to create and integrate control-plane software and basic data-plane functionality to create a complete platform.

B. Packet Processing Platforms

To address this problem, platforms that have been introduced to provide a complete solution.

VPP. The Vector Packet Processor (VPP) [9] is a layer 2-4 packet-processing stack that runs in Linux user space. It is used much like an out-of-the-box network appliance (e.g., switch, router) would be used, with a fixed data plane that can be configured through a custom command line interface or through the VPP API. VPP is built using DPDK and incorporates vector processing (batching of packets) to support high-performance packet processing. VPP is packaged with custom functions to support use cases such as a virtual switch, virtual router, gateway, firewall, and a load balancer.

Polycube. Polycube [37] is a platform built on top of eBPF technology, so the Polycube data plane runs in the Linux kernel. Polycube consists of a fixed data plane that includes IP forwarding, load balancing, filtering, and more. The data plane can be configured using a custom command-line interface.

Analysis. Both VPP and Polycube are alternatives to Linux networking in the sense that, despite either running on top of Linux (VPP) or in the Linux kernel (Polycube), there is little to no interface between each platform and the Linux packet processing pipeline. There are two major drawbacks to this. First, these pipelines are incompatible with well-established management interfaces such as netlink [40], command-line tools such as iproute2 [27], or widely-used control-plane software that expect Linux's networking interfaces, such as FRR [28]. Instead, users must utilize the bespoke management interfaces provided by the platform (e.g., pcn-iptables [36] for configuring Polycube's filtering). Second, developers building these alternative pipelines often must re-implement basic functionality, like forwarding packets based on tables, responding to ARP or ICMP packets, or control plane functionality (e.g., maintaining routing tables and choosing routes). This task can incur significant development, testing, and maintenance effort to duplicate functionality that is already available elsewhere.

In short, *today's alternative pipelines are not transparent to the rest of the system* and require applications and environments to be customized for a platform in order to access the benefits of network acceleration.

C. Hot Spots in Linux

Supporting the entire Linux networking API in a custom acceleration platform would be a daunting task. Instead of re-implementing all functionality, here we consider whether one could make minimal changes to Linux networking stack while still gaining the majority of benefits from acceleration-enabling technologies. In order to identify locations for such minimal changes we look for *hot spots* in the code which are segments that are executed frequently. We check for the existence of hot spots using flame graphs [29] for several configurations and traffic patterns. As a simple example, we configured Linux to forward traffic with *ip route* commands and observed (as seen in Figure 1) that the majority of traffic followed the same sequence of function calls — which we can then use this knowledge as a guide on what to accelerate. A key motivating observation is that Linux networking *does* have hot spots; which code section is a hot spot is dependent on networking configuration. Table I summarizes these findings for bridging, forwarding, filtering, and load balancing. For these tasks, we outline what functionality could be accelerated via a fast path to maximize performance impact, what state is accessed as part of the proposed fast path, and the functionality that should remain in the control or slow path. These observations form a basis for the design of our system.

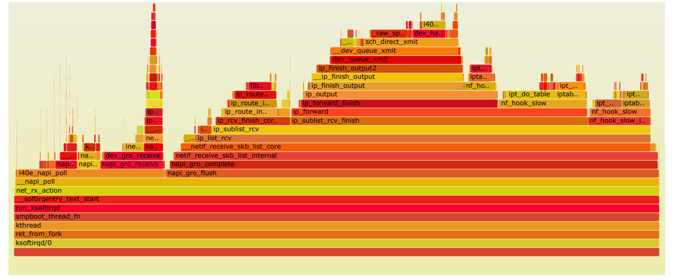

Fig. 1. Flame graph for Linux processing for forwarding.

III. Introducing LinuxFP

We introduce LinuxFP, a high-performance packet processing platform that preserves the richness of Linux networking functionality and surrounding software ecosystem. LinuxFP achieves high performance while still using most of the Linux networking stack by accelerating hot spots rather than by creating a completely new packet processing platform. Close integration between the accelerated fast path and the Linux networking slow path is a key architectural difference between LinuxFP and related work. In this section, we describe *how* and *what* LinuxFP accelerates as well as present LinuxFP's system design.

A. How to accelerate?

Dual Packet Processing Environments. Specialized packet processing environments such as DPDK and eBPF provide strong performance benefits over the traditional packet processing environment of the Linux packet processing pipeline. LinuxFP leverages both types of technology by explicitly separating out fast and slow path tasks, and then tailoring the pipeline to use each path for particular purposes. The Linux network stack is used as the slow path, as it provides complete functionality. eBPF is used to implement the fast path. The eBPF XDP and traffic control (TC) hooks within the Linux kernel enables LinuxFP to safely and dynamically load custom fast path code into the Linux kernel.

Dynamic Composability. LinuxFP accelerates identified hot spots dynamically and on demand, based on the current configuration, and only applies fast-path logic when it is needed. This is based on the general principle that less code leads to more efficient code paths, and therefore higher performance. To support this dynamism, the LinuxFP fast path LinuxFP is designed as a *composable system*, much like the models proposed in the x-kernel [31] and Click [32]. When

TABLE I
ACCELERATION MODEL FOR DIFFERENT PACKET PROCESSING APPLICATIONS.

Subsystem	Fast Path	In-Kernel State	Control Plane + Slow Path
Bridging	Parsing, rewriting, FDB lookup/update, forwarding	FDB, port state	Manage FDB (aging), handle FDB misses (flooding), STP protocol processing
Forwarding	Parsing, rewriting, FIB lookup, forwarding	FIB, neighbor tables	ARP handling, IP (de)fragmentation
Netfilter	Parsing, rewriting, conntrack lookup/update, allow/deny packets	Conntrack, ACLs	Conntrack handling, IP (de)fragmentation, handle rules on unsupported hooks
Load Balancing (ipvs)	Parsing, rewriting, conntrack lookup/update	Conntrack	Conntrack handling, Scheduling algorithms

LinuxFP uses configuration information to determine a fast path component should be deployed, it installs a series of fast path modules (FPMs). FPMs are dynamically stitched together through function calls.

Correctness Through State Sharing: Having two packet processing paths (fast and slow) means there is potential for incorrect behavior if those two paths do not share a coherent view of network state. To ensure this coherency, FPMs are designed to use dynamic (e.g., ARP table) and static state (e.g., configuration) from the Linux kernel.

B. What to accelerate?

Rule-based Hot-Spot Acceleration. We manually identified hot spots by configuring Linux in a variety of ways, generating traffic, and observing the frequently executed code for those configurations. This allowed us to create a mapping between configurations and hot spots, and then we created FPMs designed to accelerate those hot spots.

Dynamic Acceleration. We dynamically observe the configuration in order to apply rule-based hot-spot acceleration as needed. We access current configuration by inspecting Linux kernel networking state. Based on this state, we use the rule mapping from the previous step to synthesize each module in a minimal fast-path configuration and generate a packet processing graph. The graph forms the data plane that can be implemented by deploying specific FPMs. Future work includes dynamically accelerating hot spots based on traffic patterns and table accesses in addition to configuration.

C. System Overview

Figure 2 illustrates the design of LinuxFP. The LinuxFP fast path runs within the Linux kernel. For the common case, packets are only processed by the fast path. Linux is used to process corner-case packets. State is managed such that correctness is retained regardless of which path any packet takes. The LinuxFP controller runs as a daemon that continuously introspects the Linux kernel, and upon any changes will build a packet processing graph, synthesize the fast path, and deploy it. A user, for example, could enter a command with `iptables` and the LinuxFP controller would see the configuration and update the data plane accordingly (e.g., by inserting a filtering module if needed). The user is able to use their tool of choice and did not have to take any additional actions to obtain acceleration – it is incorporated into Linux transparently. This is in stark contrast to VPP and PolyCube, as summarized in Table II.

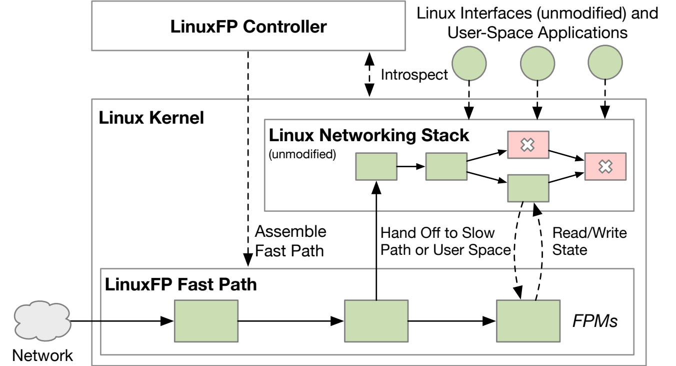

Fig. 2. LinuxFP Overview

IV. SYSTEM DESIGN

This section describes the design of LinuxFP, from the bottom up, starting with LinuxFP's dual packet processing environments. Figure 3 provides an overview of the steps required to build and deploy a LinuxFP fast path using LinuxFP fast path modules (FPMs).

A. Dual Packet Processing Environments

Most packet-processing tasks can be split into a relatively simple task for the majority of packets and one or more complex tasks a minority of packets. This insight has been used for decades in router design [25] but, despite being increasingly used for high-performance packet processing, the Linux networking stack today processes all packets in a single packet processing environment. This approach ensures any packet that is sent to Linux will be correctly processed and will interact with kernel state correctly, but it is hard to scale this approach to today's packet rates. LinuxFP aims to support dual packet processing environments: default Linux packet processing for complex/infrequent tasks, and a separate, accelerated packet processing environment defined by modules for simple/frequent tasks. FPMs are designed to only execute simple and common-case tasks so as to be fast, efficient, and reusable; corner cases and complex control protocols are left to the default Linux stack.

TABLE II
COMPARISONS OF PLATFORMS, ILLUSTRATING LINUXFP IS THE ONLY PLATFORM THAT CAN ACCOMPLISH BOTH HIGH PERFORMANCE AND TIGHT INTEGRATION WITH LINUX. *VPP WAS INITIALLY BUILT AROUND DPDK, BUT NOW SUPPORTS OTHER KERNEL-BYPASS TECHNOLOGIES (E.G., NETMAP).

	Technology	Uses Linux State	Packet Interface to Linux	Configuration API	Handling of Corner Cases
VPP	DPDK*	No	No	Custom	Custom Code
PolyCube	eBPF	No	Possible, but not used	Custom	Custom code
LinuxFP	eBPF	Yes	Yes	Linux	Linux

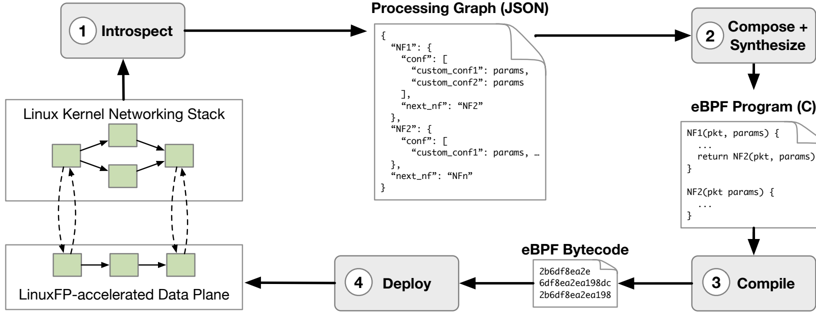

Fig. 3. Steps to install a new LinuxFP-accelerated Data Path

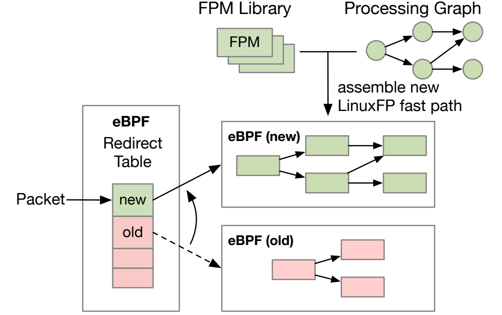

Fig. 4. Deploying a new LinuxFP Fast Path

1) Running a Fast Path within the Kernel using eBPF: Adding FPMs and switching exceptional packets from the fast to the slow path is possible using enhanced kernel programmability that was added to Linux over the past several years, namely eBPF [44] in the form of the XDP [30] and TC subsystems [6]. eBPF enables application developers to write code that can run inside a sandboxed execution environment within the Linux kernel at various locations (called *hooks*) to efficiently and safely add user-defined logic to the kernel without requiring kernel source code changes or loadable kernel modules [8].

eBPF offers two hooks for packet processing: XDP and TC. Programs attached to the *XDP hook* run on packets immediately after the packet is retrieved from the network interface card (NIC). Programs attached to the *TC hook* can run both on incoming packets and on outgoing packets before they are passed to the NIC. TC provides lower performance due to its location in the kernel's packet processing path (i.e., after socket buffer – the *sk_buff* struct in Linux – allocation and population) but, as a result, has access to a wider range of state and packet processing capabilities. LinuxFP leverages both XDP and TC hooks to realize the fast path of various Linux network functions in the form of FPMs.

2) Deploying FPMs Dynamically: Given an eBPF program that contains the desired fast path, LinuxFP deploys this program in the kernel for a given interface. Swapping the eBPF program currently deployed on either hook can incur packet loss for several seconds. To avoid this problem, the beginning of the LinuxFP fast path runs eBPF program that leverages the eBPF tail-call mechanism [7] to switch over from the old set of LinuxFP FPMs to the new one as it is simply updating a pointer versus loading a new program. Illustrated in Figure 4, each time the data path is regenerated in response to a change in network configuration, LinuxFP atomically replaces the current data path with the new one by updating the tail call reference to the new program in an eBPF map [30].

B. LinuxFP FPMs

FPMs are a central component of our design; they handle the majority of traffic, need to be synthesized on demand, and must integrate with the kernel for state access and management. This section outlines the design of these modules.

1) Module Structure: At a high level, FPMs are functions inside an eBPF program that taken together constitute an accelerated fast path. Consequently, FPMs are constrained by the capabilities of the eBPF virtual machine and eBPF hook point specific constraints. FPM code is invoked by Linux for each packet such that the program terminates after processing a single packet and any local state is discarded. Depending on the hook, different data structures are passed to the eBPF program as a pointer. For the XDP hook, since the packet has just been retrieved from the network and no other processing has been performed yet, the data available is limited to the packet buffer, the interface index, and the queue index from which the packet was received. Packets on the TC hook are associated with a populated *sk_buff* as TC eBPF programs are invoked after some initial kernel processing. The *sk_buff* contains significantly richer data than the meta data available in XDP and contains pointers to individual parsed headers, as well as data from the routing and bridging systems.

The code of each FPM is synthesized individually depending on the network configuration of the system. This code composed from a set of code snippets for individual tasks, such as performing a FIB lookup or parsing VLAN headers. As a result, branching inside the fast path (e.g., to check whether a feature such as VLANs are enabled on the device) can be reduced to a minimum as this logic is not included

if not required. This enables streamlined and efficient FPMs. Finally, depending on whether a FPM is at the end of the processing graph or not, the FPM contains code to either call the subsequent FPM or set an XDP/TC verdict that determines what happens to the packet next. Possible verdicts include dropping the packet, sending the packet to a NIC, or sending the packet to the kernel for further processing.

2) Unifying State: As FPMs implement packet processing functions such as routing and filtering on behalf of Linux, they require access to the state of the respective subsystems within Linux to maintain transparency to the rest of the system. Within our system, every packet must be able to be processed either by the LinuxFP fast path or by the kernel with the identical result under all circumstances. This is a crucial design decision and requires ensuring consistency and correctness of all operations whether they have been performed in the slow path or in the fast path. Effectively, Linux provides a superset of the functionality of the LinuxFP fast path.

This means that although there are two different packet processing environments, control plane state must be unified: an FPM must be able to view the current configuration (such as configurations set by an administrator) as well as access and sometimes modify kernel networking state. While eBPF maps can be used to share state between user space and an eBPF program, doing so would sacrifice transparency as they would have to be maintained separately; furthermore, this approach would require explicitly handling concurrent operations between user space and kernel space. Thus, instead of using maps, LinuxFP uses helper functions that can access and modify kernel state. Some of these helpers are already available in the kernel for use within eBPF programs (e.g., *bpf_fib_lookup* [20]) but other helpers (e.g., for looking up iptables entries) are not present. In those cases, we add them to the kernel as required.

3) Composing Functions inside the Fast Path: A Linux system typically performs a series of networking tasks, e.g., a firewall plus a load balancer where each task is associated with specific configuration details. As such, simply loading a single, generic FPM is insufficient. LinuxFP builds and deploys a fast path by dynamically building C code that can be compiled and loaded into the kernel to realize the current packet processing graph configured on Linux. LinuxFP models the processing graph in JSON, as illustrated in Figure 3. A code synthesizer ingests the JSON model and outputs the C code for a series of customized FPMs generated from Jinja templates [11]. To do this, the synthesizer maps each key in the model to an FPM; subkeys are used to specialize the FPM with code-sippets based on configuration details. For example, a JSON model of a bridge with STP and VLAN configured would have `bridge` as the key and {`STP_enabled: True, VLAN_enabled: True`} as the `conf` attributes in Figure 3. In this case, the bridge FPM along with snippets to parse VLAN and process STP will be added to the data path.

C. Supporting the Linux Networking APIs

A key contribution of LinuxFP is its ability to retain full compatibility with all Linux programming and configuration APIs. We achieve this by monitoring the kernel's configuration state, deriving dependencies, and assembling a data plane that is a subset of the current kernel networking stack. We now describe the steps in this process.

1) Introspecting the Linux Kernel Configuration: The Netlink protocol [40] enables exchanging messages between user space and the kernel. Our *Service Introspection* component uses Netlink to query kernel state by sending queries at controller startup to get an initial view of configured services, and also by joining multicast groups to get kernel notifications about configuration changes and updates. Received messages are converted into network object descriptions (LinuxFP *objects*) containing a type and a set of configuration attributes. For example, a *network interface object* contains the type of interface (e.g., physical or virtual), name, current state (e.g., up or down), IP configuration, and other properties.

2) Building a Processing Graph: LinuxFP models the Linux network processing configuration as a graph encoded in JSON. This model defines what network processing functions need to be included on the data path associated with a network interface, in which order, and how the processing must be customized. In this specification, the keys represent the processing nodes (FPMs) that should be included on the data path while sub-keys define custom configurations for each node as well as the next node on the data path.

Defining Processing Nodes. We add processing nodes to the JSON model by introspecting the Linux kernel configuration for different network subsystems (i.e., routing, bridging, and filtering), illustrated in Figure 3. If there are instances of those subsystems configured in Linux, we add their respective FPMs as keys to the processing graph. For example, we create a key for a bridge FPM if introspection retrieves a bridge network interface and add a key for a router FPM if L3 forwarding is enabled in the kernel (i.e., *net.ipv4.ip_forward=1*) and there are routes configured on the system.

Defining Processing Ordering. To define the ordering and processing dependencies, we follow the same ordering as packet processing happens inside the kernel. This ordering can change based on how network services are configured on Linux. For example, if there are network interfaces connected to a bridge, the bridge FPM will be the first key on their processing graph. If there are IP addresses and routes configured on a bridge interface or if there are routes pointing to a bridge subnet, a *next_nf* entry will be added to the JSON description of the bridge or router FPMs accordingly (see Figure 3). For example, a route pointing to a bridge subnet will create a *next_nf: bridge FPM* entry within the router JSON description and routes referring to the bridge interfaces will create a *next_nf: router FPM* within the bridge JSON description.

Customizing Processing Configuration. Finally, for each FPM we add sub-keys in the JSON model with specific configurations for the respective subsystem. For example,

we retrieve the configuration for all network interfaces on the system and if we have bridges configured, we get the master interface attribute for each of them, allowing LinuxFP to associate these interfaces to the bridge FPM. Similarly, LinuxFP introspects the specific configuration of each bridge configured in the kernel, with sub-keys describing if they have STP and VLAN configured.

V. IMPLEMENTATION

Controller. The LinuxFP controller is implemented in C++ and Python and consists of a series of components that work in concert to build the fast path. The first component (*Service Introspection*) introspects the Linux kernel using the Netlink protocol [40], leveraging libnl [14] (or libiptc [19] for iptables) for sending queries to the kernel and parsing responses. From these responses, we create *LinuxFP objects* which represent network services currently configured in the kernel. The objects are fed to the *Topology Manager* that derives relationships between objects and builds the *Processing Graph* (Section IV-C2). The *Fast Path Synthesizer* then uses this graph along with the library of FPMs to render templates generating the fast-path eBPF code. The *Capability Manager* ensures that the system supports the fast path being built (e.g., supports the necessary helpers) by inputting a description of available resources to the synthesizer. Finally, the *Fast Path Deployer* compiles the data plane, generating the eBPF bytecode which is deployed on the XDP or TC hooks using libbpf [13].

Decomposing Linux Network Functions. To be as efficient as possible, the LinuxFP data plane must avoid executing any unnecessary tasks. Therefore, we break down Linux processing so that the fast path executes only a few simple tasks that are sufficient to process the majority of packets. To maintain transparency to Linux, we do not maintain any state in FPMs and instead rely on state managed by the kernel. In Table I, we summarize the division between the fast and slow paths for several Linux networking subsystems, including bridging, forwarding, and filtering. Load balancing is left as future work, but we leave it in the table as another example. In general, the LinuxFP fast path performs packet manipulations, performs state lookups in the kernel, and sends control-plane messages or corner-case packets to the slow path in the kernel. For example, the fast path of a bridge is responsible for packet parsing, forwarding database (FDB) lookups (via a kernel helper), and the actual L2 forwarding. The kernel exposes FDB access and port state to the fast path via a kernel helper and performs tasks like aging, spanning tree protocol (STP), FDB miss handling, and packet flooding. Other subsystems follow the same model.

Helper Functions. To support the described decomposition in Linux, we leverage existing kernel helper functions and introduce new ones when necessary. To implement routing, we use the *bpf_fib_lookup* helper that already is available in the kernel [20] to determine the next hop for a packet by querying its destination IP in the routing table. The helper also allows setting the source MAC address of the egress interface and the destination MAC address to the next hop. For the bridge, we introduce a new helper called *bpf_fdb_lookup*; it allows querying the FDB (L2 forwarding database) to retrieve the egress port of a previously learned MAC address. The MAC learning process is delegated to the slow path. Our helper also supports FDB entry aging, VLAN filtering, and the spanning tree protocol (STP). We accelerate Linux's iptables filtering using a new helper which we call *bpf_ipt_lookup*; it enables matching source/destination IP addresses (performing longest-prefix matching) and the protocol. We also support aggregating rules using ipsets. All testing was performed with the Linux 6.6 Kernel and our helpers ($\sim$260 LoC) available in [2]; we hope to work with the community to make those helpers available in the mainstream Linux kernel.

VI. EVALUATION

We introduced LinuxFP as a means to transparently accelerate networking in Linux. As such, our first evaluation task is to determine to what degree it accelerates packet processing for common use cases (Section VI-A). Then, we present some microbenchmarks (Section VI-B) to understand the performance characteristics of implementation choices in LinuxFP.

A. Acceleration Benchmarks

LinuxFP has two main goals: acceleration, and compatibility with Linux networking APIs to enable comparison. As such, our evaluations goals are to show for some common scenarios 1) how much LinuxFP accelerates packet processing when compared to Linux, and 2) that LinuxFP does not sacrifice performance compared to other acceleration platforms. For our first scenario (two different virtual network functions), we compare LinuxFP against Linux as the baseline, Polycube (version v0.9.0) as a kernel space platform with eBPF, and VPP (version 23.10) as a user space platform with DPDK. For the second scenario (Kubernetes network plugin), we only compare against Linux as each of Polycube's and VPP's custom implemented network plugins are not perfectly comparable to our Linux setup. All experiments were run on CloudLab [23] c6525-25g hosts with 25Gbps NICs, Hyper Threading (HT) and power saving disabled, and running the Linux 6.6 kernel.

1) Virtual Network Functions: Virtual network functions are increasingly leveraged to support dynamic networks. We can see these in on-premise data centers (e.g., Equnix Network Edge [4]) and in-cloud data centers (e.g., Cisco Cloud Services Router [3]) alike. While those are proprietary solutions, we believe Linux is a suitable open alternative – if accelerated. The two network functions we consider are a virtual router and a virtual gateway.

For these experiments, we set up in a three node line topology with a traffic source and traffic sink each connected to the device under test through a separate link. We perform all tests using CPU cores and NIC interfaces on the same NUMA node. For packet generation, we leveraged DPDK's Pktgen [16] for throughput tests and netperf [15] for latency

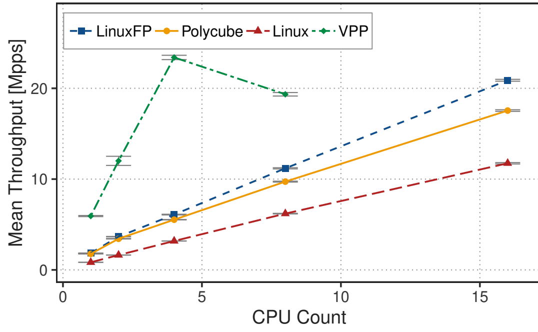

Fig. 5. Throughput of virtual router as a function of number of cores.

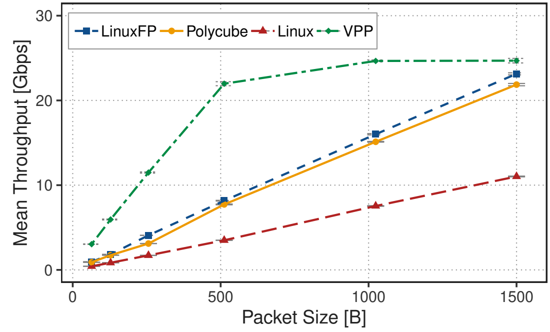

Fig. 6. Throughput of virtual router for a single core as a function of packet size.

TABLE III
VIRTUAL ROUTER RTT WITH SINGLE CORE. LATENCY MEASURED IN μS.

	Avg.	P_99	Std. Dev
Linux	326.872	512.378	109.265
Polycube	145.792	269.772	60.204
VPP	85.604	182.265	32.011
LinuxFP	151.675	279.407	76.798

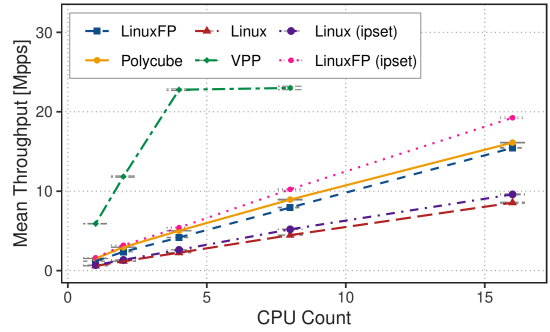

Fig. 7. Throughput of virtual gateway as a function of number of cores.

tests. We let Pktgen warm up for 10 seconds before determining the observed throughput. Unless stated otherwise, we use XDP driver mode for LinuxFP and Polycube. We run each experiment for 10 seconds and repeat it 10 times. In our tests, VPP and Polycube are configured with commands equivalent to the Linux configuration.

Virtual Router. The core function used in a virtual router is IP forwarding and virtual routers are used to interface between multiple networks. For example, in an on-premise deployment, a virtual router may route between multiple-cloud providers through direct connections. In an in-cloud deployment, a virtual router could be used to accept route advertisements from a transit gateway and forward traffic between regions.

Our first experiment uses iproute2 to set 50 prefixes and measures throughput as a function of the number of cores used. Here, we use Pktgen to send minimum sized packets. Figure 5 shows that LinuxFP's ability to generate the minimal data path for specialized routing allows it to nearly double the Linux throughput. LinuxFP and Polycube have similar performance in this use case, with differences that we attribute to implementation choices discussed later in this section (VI-B). This is significant, as LinuxFP was able to achieve similar performance while retaining the Linux networking API (while Polycube does not). VPP has higher throughput, which we attribute to its use of vector processing (batching). However, the use of busy polling (via DPDK) in VPP, requires it to dedicate the configured number of cores entirely to VPP which then run at 100% utilization. VPP uses each core to poll on a separate NIC hardware queue.

For latency, we focus on processing with a single core. Here, we load the network function using 128 parallel netperf sessions and average the results among all instances. Table III shows that LinuxFP is capable of improving the latency of Linux routing, again having performance similar to Polycube.

Finally, to exercise one aspect of scalability, we look at single core throughput for various packet sizes. Figure 6 shows LinuxFP and Polycube achieve near line rate (25Gbs in our testbed) with just one core when processing 1500B packets.

Virtual Gateway. A virtual gateway sits at the edge of networks and can both forward traffic and provide some security by enabling white listing of IP addresses or ports that can access some private service. The core functions used are IP forwarding and a network filter.

We use iptables to configure 100 rules blocking a blacklist of IP addresses for networking filtering, and iproute2 to configure 50 prefixes for IP forwarding. We first explore throughput as a function of number of cores using minimum-size packets. In Figure 7, we can see that LinuxFP nearly doubles Linux throughput for this use case. However, we inherit iptables performance issues, related to linear searches on rule tables. Polycube addresses this issue by adopting a more efficient classification algorithm [34]. Fortunately, our iptables helper implementation allows LinuxFP to leverage Linux's ipsets, which allow aggregation of several filtering rules in sets that can be matched in one or more policies. In this experiment, we aggregate the blacklist of IP addresses in one set allowing us to reduce our filtering rules to just one. This allows LinuxFP's firewall to have better performance than Polycube in this scenario. Again, VPP's performance is higher, and is included for reference despite its need for bespoke dedicated resources for packet processing.

We measure latency for packets processed on a single

TABLE IV
VIRTUAL GATEWAY RTT WITH SINGLE CORE. LATENCY MEASURED IN μS.

	Avg.	P_99	Std. Dev
Linux	388.863	512.404	40.942
Linux (ipset)	331.480	437.275	49.052
Polycube	181.500	289.379	40.584
VPP	85.604	180.948	32.011
LinuxFP	212.798	317.636	43.730
LinuxFP (ipset)	161.469	275.114	39.625

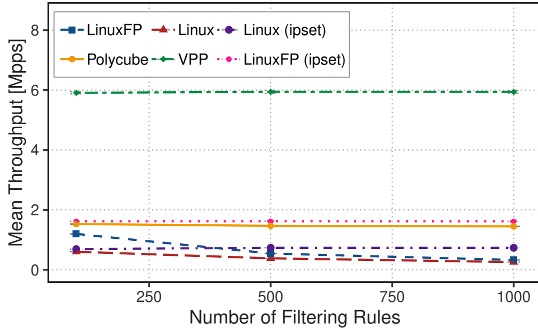

Fig. 8. Single core throughput of virtual gateway as a function of number of filtering rules.

core. Here, we load the network function using 128 parallel netperf sessions and average the results across the instances. In Table IV, we see that LinuxFP improves latency in this scenario and performs better than Polycube when using ipsets.

Finally, as a scalability test, we explore single core throughput as a function of the number of filter rules. In Figure 8 we see that LinuxFP, Linux, and Polycube scale throughput as we add CPUs. LinuxFP has better performance when aggregating the filtering rules using ipset which has also been shown to be scalable to a larger number of rules [35].

2) Kubernetes Pod-to-Pod: To understand the importance of retaining Linux Networking APIs and the power in leveraging the Linux networking ecosystem, we select Kubernetes container orchestration platform [12] as an example application to evaluate LinuxFP. Today, many networked applications are run inside containers as containers provide a powerful way to package, deploy, and scale applications in cloud environments. Containerized applications often span many containers which provide different services (e.g., caches, application logic, databases, etc.), requiring a container orchestration platform such as Kubernetes. Kubernetes enables internal communication via network plugins (e.g., Flannel [10], Calico [43], etc.) that implement the interface defined by the container network interface (CNI) specification. Network plugins often heavily rely on Linux's built-in networking capabilities such as bridging to enable communication between containers on the same host, routing and encapsulation to enable communication between hosts, address translation and load balancing for outside connectivity, and packet filtering to ensure security.

In our experiments we use the Flannel [10] network plugin with a 3-node Kubernetes cluster (one primary node and two

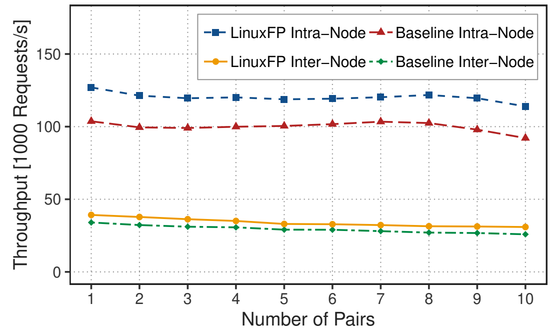

Fig. 9. Pod-to-pod communication throughput as a function of number of pod pairs.

TABLE V
POD-TO-POD LATENCY WITH SINGLE POD PAIR. MEASURED IN MS.

	Avg.	P_99	Std. Dev
Linux (intra)	9.680	20.1	2.021
LinuxFP (intra)	7.918	15.9	1.527
Linux (inter)	29.226	34.7	3.086
LinuxFP (inter)	25.176	30.9	2.913

secondary nodes). We evaluate Linux and LinuxFP by measuring the throughput and latency of pod-to-pod communication for both intra-node communication (where both pods are co-located on the same node) and inter-node communication (where the pods are located on different nodes). The LinuxFP synthesized data plane is attached to the tc hook. For both scenarios, pods are deployed in pairs, with one pod acting as a server (running netperf's netserver) and one acting as a client (running netperf [15]). For all measurements, we use the netperf TCP_RR test run for 60 seconds, which is configured to output average and 99%-ile latency as well as the standard deviation. We output the number of transmissions for each one second interval to measure throughput.

We collect throughput data for 1-10 pairs of pods that are running simultaneously (Figure 9). The first and last 10 seconds of throughput data is discarded, so per-client throughput is the average of the middle 40 seconds of data. The overall throughput is the average over all per-client throughputs of the test configuration, then this value is averaged over 10 test iterations. As shown in Figure 9, LinuxFP demonstrates 120% (intra) and 116% (inter) throughput when compared to Linux.

Latency results from the 1-pair case are shown in Table V, with each result the average of 10 test iterations. LinuxFP reduces average latency by 18% and 14% for intra and inter respectively, and reduces 99%-ile latency by 21% and 11% for intra and inter respectively.

In sum, LinuxFP shows performance improvements for both throughput and latency for both intra- and inter-node pod configurations when compared against Linux. No modifications to Kubernetes, pods, or other tooling was required to run this experiment, other than the requirements to install and run LinuxFP on each worker node.

TABLE VI
LINUXFP REACTION TIME IN SECONDS

Command	Time
ip addr add 10.10.1.1/24 dev ens1f0np0	0.602
brctl addbr br0	0.539
brctl addif br0 veth11	0.493
iptables -d 10.10.3.0/24 -A FORWARD -j DROP	1.028

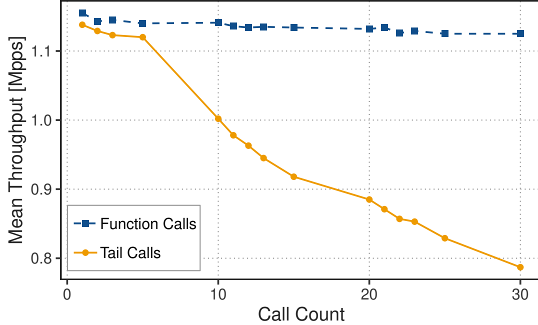

Fig. 10. Function call vs Tail call.

B. Micro benchmarks

Here, we run experiments to explore different aspects of the operation of LinuxFP.

Reaction Time. We define reaction time as the time from when a command is entered to the time of data plane installation. This includes generating the packet processing graph, synthesizing parameterized modules, and installation of the eBPF code into the kernel. We measure this time in the LinuxFP controller, where the start time is when LinuxFP sees the configuration change and the end time is when LinuxFP gets confirmation that the eBPF code is installed. The reaction time for various commands is found in Table VI.

eBPF Implementation Options. In some cases we saw better performance with LinuxFP than Polycube. As they both use eBPF, performance differences largely stem from implementation differences. One implementation factor is how functions are chained together: LinuxFP chains functions with inlined function calls, while Polycube chains functions together with tail calls. While a greater study of how best to use eBPF is beyond the scope of this paper, and even further optimizations likely exist, we did craft an experiment independent to highlight how these implementation differences may impact performance. In this experiment, which was independent of either platform, we created a chain of N trivial network functions, followed by one function that modifies the Ethernet and IP headers and then uses XDP_REDIRECT to forward the packet out of another interface. Varying the number of functions allows us to see the impact of using a tail call vs. a function call. We use the same experimental setup as in the Virtual Network Functions use case benchmark. In Figure 10, we can see that the throughput remains relatively steady when using function calls, but drops by about one percent for each added function when using tail calls.

TC Hook vs. XDP Hook. The LinuxFP controller can insert the needed function into either the TC hook point or the XDP hook point. Here, we explore the impact of that choice by measuring the throughput. This is important, as depending on the use case, XDP or TC is more appropriate. For example, in a container scenario, where containers will consume packets, allocating the *sk_buff* is inevitable. In this case, a TC data plane performs better as we can, for example, avoid the cost of converting the *xdp_buff* into a *sk_buff*. In Table VII, we evaluate several network functions in a forwarding scenario and compare TC and XDP dataplanes. In this case, we can see that XDP performs better as we can avoid the costs of *sk_buff* allocation and other unnecessary processing steps, as we process the packet closer to the wire.

TABLE VII
LATENCY AND THROUGHPUT OF FUNCTIONS USING XDP VS TC HOOKS.

	Throughput (pps)		Mean Latency (ms)	
	XDP	TC	XDP	TC
bridge	1,914,978	889,735	139.523	275.300
forwarding	1,768,221	850,209	149.248	288.139
filtering	1,183,252	680,065	215.611	363.133

VII. RELATED WORK

This paper extends an earlier workshop paper [1] which accelerated only one Linux network subsystem (bridging). LinuxFP goes beyond that work by building a complete controller that can accelerate several Linux networking subsystems, evaluating each individually and in combinations, as well as evaluating the more complex use case of a Kubernetes network plugin. Beyond that, this paper shares similarities with a few bodies of related work.

Kernel-bypass networking. A variety of packet I/O frameworks take the approach of bypassing the kernel in order to scale software packet processing, most notably the Data Plane Development Kit [21], PF_RING [41], and Netmap [42]. Common to these frameworks is that they generally take over control of a NIC, only copy packets a single time from the NIC to pre-allocated memory via DMA, and rely on expensive busy polling instead of interrupts. In contrast, with LinuxFP we believe that the Linux networking stack should not be bypassed, but instead redesigned such that we can leverage the operating system's networking features, and its ecosystem of tools and control plane software.

In-kernel accelerated packet processing. There has been work that can load custom packet processing functionality into the kernel, providing both the opportunity to access kernel state (e.g., the forwarding table) and exchange traffic with the Linux networking stack. Click [32], eBPF / XDP [30], and VPP [9] are described in more detail in Section II. LinuxFP is complementary to these efforts, as we rely on the capabilities of eBPF / XDP to provide a fast path packet procressing environment and are inspired by the composibility of Click modules. Bastion [39] implements a CNI with XDP has a smaller scope than LinuxFP, as LinuxFP is focused on Linux networking, generally.

Clean-slate approaches. Finally, entirely new kernel architectures have also been proposed. X-kernel [31] is an early work that proposes an OS designed to simplify building and composing communication protocols; it includes abstractions and building blocks to realize a wide range of protocols to be used within and across hosts. More recently, Zhang et al. proposed Demikernel [45], an OS architecture that aims at integrating legacy control plane software with a fast data path bypassing the kernel. Those approaches have in common that they propose completely new kernels and radical changes to OS architecture. While achieving similar goals, LinuxFP can be deployed today as it can be implemented using mechanisms already provided by Linux.

VIII. Conclusions and Future Work

In this paper, we introduce LinuxFP, a system which takes a unique look at incorporating advances in high-performance packet processing into Linux. In doing so, we accelerate Linux's network stack completely transparently to common command-line tools such as brctl and iptables, control plane software such as FRR, and management frameworks such as Kubernetes and Ansible. We leverage eBPF programmability in the Linux kernel to enable deploying dynamically synthesized data planes based on current configuration. Evaluations with our implementation of LinuxFP showed that it provides 77% higher throughput and 53% lower latency than Linux. We also showed that LinuxFP does not sacrifice performance in comparison to Polycube (an alternative pipeline that also uses eBPF) while maintaning full support for Linux network APIs and tight integration with Linux packet processing. We also evaluate the throughput of pod-to-pod communication in Kubernetes, and find that with LinuxFP, performance increases 20% as compared to without LinuxFP (i.e., standard Linux).

For future work, we intend to pursue several opportunities. First, while we have accelerated the bridging, routing, and filtering functionality within Linux, there is other functionality that can be accelerated. We have begun work on ipvs, the load balancing functionality within Linux (and used in Kubernetes services), and initial prototyping is showing promising results. In addition to accelerating Linux functions, we intend to support the insertion of custom functionality, e.g., for monitoring [18] modules. We can inject custom eBPF code at different points in the XDP processing pipeline or add custom packet-processing applications in user space [17] and use a special type of socket, called *AF_XDP*, that allows sending raw packets directly from the XDP layer to user space. Finally, the architectural decision to incorporate the concept of an explicit fast path (as is done in network systems already) lends itself nicely to hardware offload. We intend to explore this in the context of SmartNICs and FPGAs.

Acknowledgements

We thank the ICDCS reviewers for their helpful feedback on this work. We would also like to thank Chethan Kavaraganahalli Prasanna and Akshay Abhyankar for their contributions to this project through early prototypes. This work has been supported in part by NSF as part of CNS Core award no. 2241818, a Google PhD fellowship, DARPA grant HR0011-20-C-0107, and the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001.

References

[1] Anonymized.
[2] Author's Linux kernel fork - new BPF helper implementations. https://github.com/mcabranches/linux.
[3] Cisco Cloud Services Router (CSR) 1000V. https://aws.amazon.com/marketplace/pp/prodview-b75wijpubtr3k.
[4] Equinix network edge. https://www.equinix.com/products/digital-infrastructure-services/network-edge.
[5] Aviatrix multi-region high-availability cloud network design, 2020. Retrieved April 28, 2023, from https://aviatrix.com/wp-content/uploads/2020/09/Aviatrix-Validated-Design-Multi-Region-HA-2.pdf.
[6] tc-bpf(8) — linux manual page, 2022. Retrieved April 29, 2023, from https://man7.org/linux/man-pages/man8/tc-bpf.8.html.
[7] Cilium bpf and xdp reference guide: Tail calls, 2023. Retrieved April 27, 2023, from https://docs.cilium.io/en/stable/bpf/architecture/#tail-calls.
[8] ebpf documentation: What is ebpf?, 2023. Retrieved April 29, 2023, from https://ebpf.io/what-is-ebpf.
[9] FD.io: the worldś secure networking data plane, 2023. Retrieved October 20, 2023, https://fd.io.
[10] Flannel, 2023. Retrieved May 1, 2023, from https://github.com/flannel-io/flannel.
[11] Jinja, 2023. Retrieved May 1, 2023, from https://jinja.palletsprojects.com/.
[12] Kubernetes, 2023. Retrieved May 2, 2023, from https://kubernetes.io.
[13] Libbpf, 2023. Retrieved May 1, 2023, https://libbpf.readthedocs.io/en/latest/index.html.
[14] Netlink protocol library suite (libnl), 2023. Retrieved May 1, 2023, https://www.infradead.org/ tgr/libnl/.
[15] Netperf, 2023. Retrieved May 1, 2023, from https://github.com/HewlettPackard/netperf.
[16] Pktgen - traffic generator powered by dpdk, 2024. Retrieved Jan 19, 2024, from https://github.com/pktgen/Pktgen-DPDK.
[17] Marcelo Abranches and Eric Keller. A userspace transport stack doesn't have to mean losing linux processing. In *IEEE NFV-SDN*. IEEE, 2020.
[18] Marcelo Abranches, Oliver Michel, Eric Keller, and Stefan Schmid. Efficient network monitoring applications in the kernel with eBPF and XDP. In *IEEE NFV-SDN*. IEEE, 2021.
[19] Leonardo Balliache. Querying libiptc howto, 2022. Retrieved October 24, 2022, from https://tldp.org/HOWTO/Querying-libiptc-HOWTO/index.html.
[20] Linux, bpf-helpers(7) — Linux manual page. https://man7.org/linux/man-pages/man7/bpf-helpers.7.html, 2021. Retrieved June 10, 2022.
[21] DPDK Project. Data plane development kit, 2022. Retrieved June 13, 2022, from https://www.dpdk.org.
[22] P. DRUSCHEL, L. PETERSON, and B. DAVIE. Experiences with a high-speed network adaptor: A software perspective. In *ACM SIGCOMM Conference (SIGCOMM)*, 1994.
[23] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.
[24] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *USENIX NSDI*, 2016.
[25] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to sdn: An intellectual history of programmable networks. *SIGCOMM Comput. Commun. Rev.*, 44(2):87–98, apr 2014.
[26] Open Networking Foundation. Aether, 2022. Retrieved June 16, 2022, from https://opennetworking.org/aether.

[27] The Linux Foundation. iproute2, 2022. Retrieved June 21, 2022, from https://wiki.linuxfoundation.org/networking/iproute2.

[28] FRR Project. FRRouting project, 2022. Retrieved June 14, 2022, from https://frrouting.org.

[29] Brendan Gregg. Flame Graphs, 2020. Retrieved Jan 19, 2024, from https://www.brendangregg.com/flamegraphs.html.

[30] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: Fast programmable packet processing in the operating system kernel. In *ACM CoNEXT*, 2018.

[31] N. Hutchinson and L. Peterson. Design of the x-kernel. In *ACM SIGCOMM*, 1988.

[32] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *Transactions on Computer Systems*, 18(3), aug 2000.

[33] Teemu Koponen et al. Network virtualization in multi-tenant datacenters. In *USENIX NSDI*, 2014.

[34] TV Lakshman and Dimitrios Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. *ACM SIGCOMM Computer Communication Review*, 28(4):203–214, 1998.

[35] Praveen Likhar and Ravi Shankar Yadav. Impacts of replace venerable iptables and embrace nftables in a new futuristic linux firewall framework. In *2021 5th International Conference on Computing Methodologies and Communication (ICCMC)*, 2021.

[36] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Mauricio Vásquez Bernal, Yunsong Lu, and Jianwen Pi. Securing linux with a faster and scalable iptables. *ACM SIGCOMM Computer Communication Review*, 49(3):2–17, 2019.

[37] Sebastiano Miano, Fulvio Risso, Mauricio Vásquez Bernal, Matteo Bertrone, and Yunsong Lu. A framework for ebpf-based network functions in an era of microservices. *IEEE TNSM*, 18(1), 2021.

[38] J. C. MOGUL and K. K. RAMAKRISHNAN. Eliminating receive livelock in an interruptdriven kernel. *ACM Trans. Computer Systems*, 15:217–252, 1997.

[39] Jaehyun Nam, Seungsoo Lee, Hyunmin Seo, Phil Porras, Vinod Yegneswaran, and Seungwon Shin. BASTION: A security enforcement network stack for container networks. In *USENIX ATC*, July 2020.

[40] netlink(7) — Linux manual page. https://man7.org/linux/man-pages/man7/netlink.7.html, 2021. Retrieved June 10, 2022.

[41] NTOP. PF_RING: High-speed packet capture, filtering and analysis, 2022. Retrieved June 13, 2022, from https://www.ntop.org/products/packet-capture/pf_ring.

[42] Luigi Rizzo. netmap: a novel framework for fast packet i/o. In *USENIX ATC*, 2012.

[43] Tigera, Inc. Project calico, 2022. Retrieved June 15, 2022, from https://www.tigera.io/project-calico.

[44] Marcos A. M. Vieira, Matheus S. Castanho, Racyus D. G. Pacífico, Elerson R. S. Santos, Eduardo P. M. Câmara Júnior, and Luiz F. M. Vieira. Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Comput. Surv.*, 53(1), feb 2020.

[45] Irene Zhang et al. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *ACM SOSP*, 2021.