Getting back what was lost in the era of high-speed software packet processing

Marcelo Abranches
University of Colorado Boulder
made0661@colorado.edu

Oliver Michel
Princeton University
omichel@princeton.edu

Eric Keller
University of Colorado Boulder
eric.keller@colorado.edu

Abstract

The need for high performance and custom software-based packet processing has resulted in decades of research. Most proposals bypass or replace the Linux networking stack with the unfortunate consequence of sacrificing the rich and robust functionality available within Linux and the ecosystem of management programs and control-plane software built on top of it. In this paper, we propose to rethink the design of the Linux network stack to address its shortcomings rather than creating alternative pipelines. This re-design involves (1) decomposing packet processing into a fast path and a slow path, and (2) transparently and dynamically creating a custom fast path that only implements the processing tasks currently configured. We leverage Linux's eXpress Data Path to load efficient and small fast-path modules, leaving the kernel stack to serve as the slow path. To materialize this vision, this paper introduces Transparent Network Acceleration (TNA), a prototype system that automatically generates a minimal data path based on introspection of the current networking configuration, avoiding many of the networking stack overheads in Linux while ensuring high performance and maintaining Linux's rich set of functionalities.

CCS Concepts

• **Networks** → **Programmable networks**;

Keywords

Middlexboxes, NFV, Network Management, Network Stacks.

ACM Reference Format:
Marcelo Abranches, Oliver Michel, and Eric Keller. 2022. Getting back what was lost in the era of high-speed software packet processing. In *The 21st ACM Workshop on Hot Topics in Networks (HotNets '22), November 14–15, 2022, Austin, TX, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3563766.3564114

1 Introduction

Software-based packet processing is being widely adopted across a number of use cases, such as data center load balancing [8], virtualized networking between containers [26, 27] or virtual machines [16], and in 5G infrastructures [9]. Doing so 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 services, supporting the packet processing performance required with its out-of-the-box network stack is challenging [5, 12, 14].

This led to new approaches for enabling high-performance custom packet processing through alternative pipelines. These take several different forms, such as kernel bypass (e.g., DPDK [7], netmap [23]) which efficiently copy packets to a user space program for processing, in-kernel network stack bypass (e.g., XDP [12], Click [15]) which run inside kernel space but are still largely an alternative pipeline as performance is only obtained when the traffic does not touch the Linux network stack, or as a new kernel (e.g., x-kernel [13], Demikernel [28]).

Using an alternative pipeline vastly improves throughput over receiving and processing packets through the Linux networking stack and makes it easier to add new functionality; however, it incurs significant drawbacks. First, these alternative pipelines cannot, without degrading performance, leverage the rich and widely used networking functionality of Linux such as its built-in bridging, packet filtering, and traffic shaping subsystems together with their powerful management tools (e.g., iproute2 [10]). The rich ecosystem extends to management software that builds around the Linux APIs and interfaces (and command line tools), such as Infrastructure-as-Code (e.g., Ansible), container networking (e.g., Flannel), and control plane software (e.g., FRR [11] for routing and StrongSwan [25] for IPsec).

In this paper we take the position that we should rethink the design of the Linux network stack to address its shortcomings, rather than creating alternative pipelines. Moreover, we show that this is practical today.

Overheads in the Linux networking stack. To understand what is needed as part of a re-design, we need to look at the overheads in the Linux networking stack. The generality that makes Linux networking so powerful is also one of its main sources of inefficiencies. One dimension of this is that there is a long, complex data path that performs too many operations per packet. This includes many different protocols and a wide array of functionality where Linux needs to check whether each block of code to process needs to be called or not. This leads to a long critical path which, in turn, slows processing. A second dimension is that the processing that does need to be executed is quite inefficient — again, due to the need for generality. This includes both heavyweight protocol implementations that can support all of the corner cases (e.g., IP fragmentation), as well as heavyweight data structures (e.g., *sk_buff*) for which the allocation process is not just a memory allocation, but complete parsing of packets to fill in all of the data into the structure.

There are two key insights that we can draw from this — both opportunities for optimization of Linux networking. First, in most cases, only a subset of functionality is actually being used. For example, we might only need to set up a bridge between two interfaces, but Linux still checks if we are using IPsec, packet filters, or traffic shaping. Instead, we believe Linux should be designed much like the models proposed in the x-kernel [13] and Click [15] work — that is as a *composable design*. However, unlike those in which users explicitly provide a graph of processing, we need to make this transparent from users of Linux.

Second, while Linux does have a degree of fast-path and slow-path processing, such as the inclusion of some control protocols (e.g., spanning tree) in the kernel, it treats all packets the same — with the same pipeline and same data structures being allocated. Instead, we believe Linux should explicitly separate out fast-path functionality and slow-path functionality, and tailor the execution of each.

Rethinking the Linux networking stack is practical. Redesigning the Linux networking stack as proposed, would require (1) a decomposition into explicit fast-path and slow-path functionality and execution environments, and (2) an ability to dynamically instantiate only the part of the network stack that is used. While this is counter to the monolithic design of Linux, we believe that, while not explicitly designed for this purpose, there are existing frameworks that can serve as the fast-path execution environment. The key things we need in an execution environment for fast-path processing are (1) that it is designed with efficiency in mind for high throughput processing, (2) that it enables the dynamic loading of processing so that we can load only what is needed at that particular time, and (3) an ability to interact with the Linux kernel to be able to access its data structures and exchange packets. Both Click and, more recently, XDP provide all of these. What having this fast-path execution environment enables is Linux, as exists today, to serve as the slow path. With that, two more challenges remain: (1) how to design the modules such that they are very lightweight and leverage the current Linux network stack as the slow path, and (2) how to dynamically build a graph of Linux processing of what is needed.

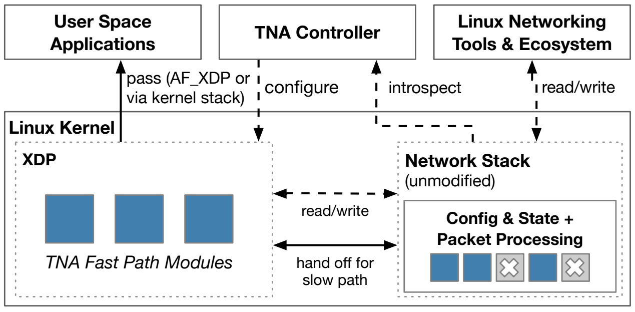

Figure 1: TNA Overview.

Introducing TNA. As a prototype realization of this vision, we introduce *Transparent Network Acceleration (TNA)*, shown in Figure 1, which includes a library of fast-path modules of various Linux networking functions coupled with an orchestration layer that builds a custom data plane on demand based only on what is configured in Linux (with command line utilities or other software that use the Linux interface to set data structures within the Linux kernel, such as the forwarding table). The modules are designed to be lightweight and can be stitched together and loaded into the kernel with XDP. They are designed to work with the Linux networking stack as the slow path by redirecting any need for corner case processing or state management to the Linux networking stack and accessing state through various helper functions. Also, we support different application needs by multiplexing packets to custom processing based on packet header information. To determine what fast-path processing graph to load at any given time, a lightweight controller continuously introspects the kernel's network configuration, then it composes and places the currently used functions on the data path. It also provides APIs for user-defined logic to be inserted at any point in the data path. The result is a fast Linux networking stack that is (1) *fast* as it only runs a slim data plane of the currently required functions instead of passing packets through a complex, general-purpose stack, (2) *transparent* to the rest of the system as it uses the kernel's configuration and network state together with its ecosystem of tools and third-party software, and (3) *extensible* as it uses an underlying technology (XDP) that was designed to add custom functionality to the Linux kernel efficiently.

To demonstrate the benefits and performance gains with TNA, we accelerate the Linux bridging subsystem. With this, the user can configure Linux bridges with the brctl utility and TNA transparently accelerates the packet processing. Depending on the CPU count, the TNA-accelerated bridge improves throughput by 3.5–4.5 times over the Linux networking stack implementation. We also compare with Polycube [17], which directly leveraged XDP to accelerate bridging by bypassing the Linux networking stack. Unlike TNA, Polycube is not transparent to the user and requires users to interact with custom control software. We show that even in this case, the TNA-accelerated bridge is 1.5–2.5 times faster. This largely stems from XDP being used as a bypass of the Linux networking stack in Polycube, versus an explicit leveraging of Linux functionality as a slow path that keeps the TNA modules very lightweight.

In the remainder of the paper, we first discuss how we can decompose Linux functionality into fast-path and slow-path elements (Section 2) and how we can then automatically build a custom fast data path that only includes needed functionality based on the current context (Section 3). We then describe the prototype implementation of TNA, along with an evaluation based on accelerating the Linux bridge subsystem (Section 4). We discuss related work in Section 5 and conclude in Section 6.

2 Building Composable Fast-Path Modules

The first challenge to address is how to design the modules such that they are very lightweight and leverage the current Linux network stack as the slow path effectively. Here, we first provide guidance on how to design the modules and then describe how we decompose Linux functionality into fast-path modules.

Designing fast-path modules. The Linux networking stack today processes all packets without explicitly distinguishing them as fast-path or slow-path. In this way, any packet that is sent to Linux will be correctly processed and will correctly update internal state. In the trivial case, we can consider that a fast path that is empty and sends all traffic to the Linux networking stack will work correctly.

What we aim for is that elements be inserted such that they can process a majority of packets without needing to send them to the Linux networking stack. These elements should only execute a few simple tasks such that they can be as fast as possible. What this means is different for each function; we provide some examples later in this subsection. In general, it should only include the common-case data plane processing — with corner cases and more complex control protocols being handled by Linux. In XDP, this is enabled through the ability to inject packets into the Linux networking stack. So, modules must include functionality that determines whether a packet can be processed just in the fast path, or needs to be passed to the Linux stack.

The functions should not maintain their own state, but instead only use existing Linux networking data structures. In XDP, this can be done through the use of helper functions [1]. This allows any given packet to be processed by the Linux networking stack without modification, otherwise synchronization would be needed. State will largely be (but does not need to be) read-only as much of the state management exists within the higher-level control protocols or set from management utilities (e.g., Linux command line tools) or user-space control plane software.

[1]Some kernel helpers are available to XDP today (e.g., *bpf_fib_lookup* [3]), but some are missing. So, to realize our full vision, we will both leverage the ones that are available and build new ones as needed.

Building a library of composable data-plane modules. To provide examples of these principles in practice, we elaborate in Table 1 how we break Linux packet processing into a series of lightweight modules for a set of Linux networking subsystems. While this is not an exhaustive set of networking functionality, this list is highly representative as those services serve as the basic building blocks to implement complex networking applications such as bridges, routers, NAT, firewalls, and container network interfaces (CNIs).

The lightweight modules for each subsystem are responsible for simple tasks, like parsing and rewriting packets, looking up state in the kernel tables, and sending packets for full processing when needed. We call those lightweight modules *TNA fast-path modules (FPMs)* and we build a library of them to execute tasks needed by each network subsystem. For example, the fast path on a bridge deployment can be composed by a series of pre-built TNA FPMs where each of them performs tasks like packet parsing and rewriting, forwarding database (FDB) lookups (via a kernel helper), and L2 forwarding (directly from the fast path). The Linux kernel exposes FDB access and port state to the fast path via a kernel helper and performs tasks like aging, spanning tree protocol (STP) and FDB misses handling, and packet flooding. FDB misses/flooding should happen only for the first packet destined to an unknown MAC address [4]. In this manner, the fast path is able to process the majority of the packets with higher performance than the full Linux processing (see Section 4).

In the same manner, the fast path for a router uses TNA FPM modules to parse packets, perform FIB (forwarding information base) lookups, and L3 forwarding. The fast path gets assistance from Linux by accessing exposed FIB and neighbor data via a helper, supporting routing protocols like BGP (in which protocol messages only need to be processed every few seconds [22]), and other unsupported operations like handling fragmented packets (which can be avoided [6]).

The netfilter subsystem exposes access control lists (ACLs) and the table of initiated L4 connections/flows (called conntrack table) to the fast path. This allows building accelerated services like stateful firewalls and NAT[2].

[2]Note that this model is essentially different from solutions like [17] which maintains state in BPF maps and reimplements several networking features in BPF form, missing the opportunity to leverage features already implemented in the Linux ecosystem.

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
Routing	Parsing, rewriting, FIB lookup, forwarding	FIB, neighbor tables	Routing Protocols (e.g., BGP, OSPF), 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

Table 1: Acceleration model for different packet processing applications.

3 Automated Fast-Path Data Plane Creation

The second key challenge is how to dynamically build and load a fast-path processing graph consisting of only what is needed. Before providing a more complete description, to give intuition on how TNA can generate and deploy a minimal XDP fast path to accelerate Linux networking services, in agreement with what is currently configured, we describe a simplified example for one use case. In this example, we show how TNA would automatically instantiate an XDP fast path to transparently accelerate a bridge deployment. The process starts with an operator or automation framework (e.g., Ansible) executing a sequence of commands using Linux configuration tools (e.g., iproute2 and brctl) to configure the bridge. So, the operator adds a network interface on the Linux system (NIC1) to a desired VLAN (Y). The next step is to issue a command to create the bridge (br1). After that, NIC1 is added to br1 and STP is enabled on this bridge.

To automatically generate the XDP fast path for this example, we need to *introspect the Linux kernel* looking for bridge objects and retrieve the entities that compose them (TNA objects). After that, we *build a dependency graph* with relationships between those objects, allowing us to have a structured view of the current bridge configuration. The next step is to map the nodes of the graph into one or more small units of pre-build eBPF code (TNA FPMs) that can be stitched together to provide the necessary fast-path logic to accelerate the bridge. In this example, the dependency graph contains a bridge with VLANs and STP configured on it. In this case, we *stitch together and deploy a set of TNA FPMs* on NIC1 to (1) parse L2 header including VLANs, (2) send STP messages for kernel processing (slow path), (3) perform FDB lookups in the kernel, using a kernel helper, (4) when a lookup fails, send the packet to the kernel, (5) when the lookup succeeds, send the packet to the egress port directly from the fast path.

We design the TNA controller (shown in Figure 2) with a series of components that work together to allow the steps just described to happen, as we explain next.

Introspect the Linux kernel. Our *service introspection* component uses the Netlink protocol [19] by both sending queries to the kernel at the controller startup time to get an initial view of the current configured services and also by

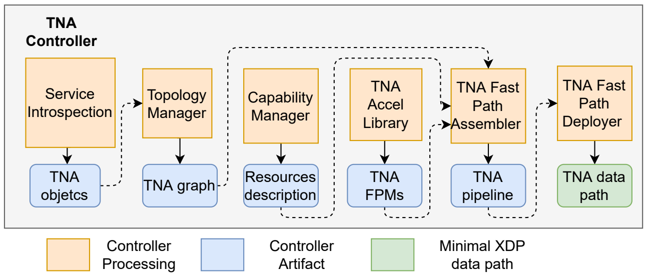

Figure 2: TNA Controller

joining to multicast groups to get kernel notifications about configuration changes and updates. The received messages are converted into network objects descriptions (TNA objects) containing the type of object and a set of configuration attributes. A network interface, for example, will contain the type of interface (e.g., physical or virtual), its name, its current state (e.g., up or down), IP configuration, and so on.

Build a dependency graph. TNA starts this process by feeding the TNA objects generated by the *service introspection* to the *topology manager* component, which is responsible for processing each of those objects and establishing relationships among them. This creates the *TNA graph*, which is a dependency graph representing the services that are currently configured and the objects that compose them. To build the dependency graph, we leverage relationships that can be derived from the Netlink messages (e.g., the bridge that an interface is part of). Where this direct mapping is not possible, we apply domain knowledge to derive other dependencies. [3] For example, when we have an IP address configured on a bridge interface, packets arriving at this interface may need routing, so we add the required objects representing this feature to the dependency graph accordingly.

TNA aims to allow tailoring acceleration code for systems according to the features they have available, such as the presence of SmartNICs with XDP offload capabilities [20] and a kernel version with needed helpers available. For doing so, the *capabilities manager* component builds an inventory

[3]Currently, we do hard code domain knowledge in TNA. As future work we aim to generalize defining supported kernel objects, services, configurations and dependencies via configuration files.

of the available assets on a given system, such as the kernel version, and network interface model.

Stitch together and deploy a set of TNA FPMs. The *TNA acceleration library* has a set of pre-built TNA FPMs. The *fast-path assembler* component uses outputs from the topology manager, capability manager, and the TNA acceleration library to generate a minimal fast path to support the services that are currently configured on the system in accordance to its capabilities. Currently, TNA does so by directly mapping one or more TNA FPMs to each node of the TNA graph in a hierarchical way that allows building a fast path that has functional equivalence with the original Linux data path (but is thinner and faster). We aim to generate a data plane that is as thin as possible as this results in fewer instructions per packet (making processing faster), potentially reduces cache misses (as code/data are more likely to fit on processor's cache), and leads to less resource consumption — which is important on SmartNIC XDP offloads. Generating a minimal data plane can be enforced by (1) avoiding code duplication by carefully organizing the TNA graph nodes such that several services on a pipeline can share TNA FPMs (e.g., a packet parser) and (2) avoiding deploying unnecessary code. For example, if there are no VLANs or IPv6 configured on the system, TNA will not deploy TNA FPMs to parse those protocols.

Given the minimal fast path that was just generated, the *Fast Path Deployer* is responsible to actually deploy the code on XDP. Currently, we do this by having, at the beginning of the pipeline, an XDP program that leverages the eBPF tail call mechanism to send packets to the minimal data path. Each time the data path is regenerated (which is triggered by changes in configuration), TNA atomically replaces the current XDP data path with the new one, by updating the tail call reference to the new program on the eBPF map [12]. The XDP data path is built by in-lining the required FPM modules to compose the full processing pipeline.

Extensible Fast Path. While the focus of this paper is how to redesign the Linux networking stack itself to be high performance, we of course inherit the desire for users to add custom packet processing. For this, TNA provides an API that allows injecting custom code on the packet processing pipeline. This can be done in two ways. The first is to inject custom eBPF code at different points in the XDP processing pipeline. Those attachment points can be at the beginning of the processing, at the end (e.g., before a packet is forwarded), or somewhere in between (i.e., between two TNA FPMs). This allows injecting custom network functionality in the pipeline, e.g., monitoring [2], or load balancing [24], that can work in concert with the deployed pipeline. The second possibility is to add custom packet processing applications on user space (e.g., [1]). This can be done by using a special type of socket, called *AF_XDP*, that allows sending raw packets directly from the XDP layer. This enables creating a hybrid kernel/user space processing environment where lower layer protocol processing and security are provided by Linux/TNA and upper layer processing (e.g., L4-L7) is provided by user space with higher performance than a full Linux pipeline. Full exploration is left as future work.

4 Prototype and Evaluation

We built an initial prototype to evaluate TNA's feasibility and performance. For our initial evaluation, we focus on the use case of accelerating Linux' bridge subsystem; it has several mature features implemented in the kernel (e.g., STP handling and MAC learning) and is widely deployed, for example in data center networks and CNIs.

TNA bridge prototype. As there currently is no helper function available in XDP to interact with bridge state inside the kernel, we needed to add one. When a packet arrives at the XDP layer, the helper adds the packet's source MAC address/ingress port to the kernel FDB table (if not yet present). After that, given a destination MAC address/VLAN ID, if there is a match on the FDB, the helper answers with the output port and also the STP state (e.g., blocked or learning).

The TNA controller is able to introspect the kernel, build the dependency graph representing the kernel objects required for a bridge, including the bridge name, the attached network interfaces, their configuration (e.g., VLANs, STP, etc.). Based on this graph, the TNA fast-path assembler composes a minimal data path. For example, if there are no external routes configured on a system, TNA will not be accelerating L3 forwarding nor will code to parse the IP header be added.

Evaluation. We now evaluate TNA's ability to transparently accelerate a bridge deployment. To do so, we set up a testbed composed of three servers. Two of them act as packet generators using DPDK's Pktgen. Both servers use a 10 Gbps Intel NIC, and we use one CPU core to generate line rate traffic (with 200 different MAC addresses) in one direction at minimum packet size (close to 15 Mpps). Traffic in the opposite direction is generated at 15 Kpps to keep FDB tables "warm". The third server runs the bridge deployment and is used to forward traffic between the other machines using 10 Gbps Intel NICs directly connected to the packet generators; we disable *hyper threading* and *power saving*. We run each experiment 10 times for 10 seconds.

We deploy and compare TNA with both Linux (kernel 5.15) and Polycube [17] (v0.9.0). As Polycube completely reimplements the bridge in eBPF/XDP and user space, it cannot be configured with standard Linux tools; the command *polykubectl* is required instead. In contrast, we configure the Linux bridge with standard unmodified tools (e.g., *ip*, *brctl*, *bridge*), and let the TNA controller automatically deploy an accelerated XDP data path for this deployment. Figure 3

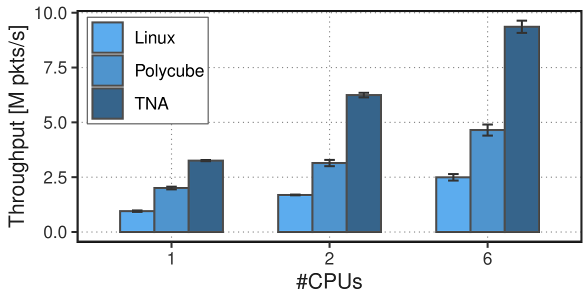

Figure 3: Throughput of Bridge Implementations.

shows the results using 1, 2, and 6 cores ((Intel Xeon E5-2620 v2 @ 2.10GHz) for each system. We can see that the Linux bridge with TNA acceleration is up to 4.5 times faster than without. It is also up to 2.5 times faster than the Polycube bridge, with the added benefit of leveraging the Linux bridge implementation and the respective configuration tools. In this scenario, TNA is faster due to its ability to generate a minimal data plane (avoiding unnecessary packet processing overheads) and optimized access to kernel data structures, which allows it to consume 38% and 64% fewer cycles per packet than Polycube and Linux respectively.

5 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 [7], PF_RING [21], and Netmap [23]. 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 TNA 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 fast 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. One such framework is the Click [15] modular router, which allows stitching together packet processing elements as a directed graph to build complex network functions from an extensive library of elements and loading that into the kernel as a kernel module. The eXpress Data Path (XDP) [12] similarly provides an in-kernel execution environment, but provides better safety through the use of the eBPF virtual machine, and has been integrated into mainline Linux. TNA is complementary to these efforts, as we rely on the capabilities of XDP to provide a fast path execution environment and are inspired by the model in Click where modules can be stitched together. As example applications built with XDP, most related to our work are Polycube (which implemented alternate implementations of some Linux network functions with XDP along with custom interfaces) and Bastion [18] (which implements a CNI with XDP). While providing acceleration, these fall short in that they bypass the Linux networking stack and, in the case of Polycube, slow-path processing needed to be implemented from scratch in user space. In contrast, TNA uses Linux' existing rich functionality as the slow path, avoiding the need for costly reimplementation and non-standard interfaces.

Clean-slate approaches. Finally, entirely new kernel architectures have also been proposed. X-kernel [13] 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 [28], 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, TNA can be deployed today as it can be implemented using mechanisms Linux already provides.

6 Conclusion and Future Work

In this paper, we propose a redesign of the Linux network stack, making it more suitable to address the needs of modern network systems in terms of performance, functionality and extensibility. The redesign starts from the observation that it is possible to instantiate a fast data path to Linux, covering only functionality that is actually in use on the system, avoiding many overheads that slow down Linux packet processing. This can be achieved with technology that is currently available in the Linux kernel. We show that our prototype is 4.5 times faster than Linux for the bridge use case. There is still work to do to realize our vision. First, we need to do a more comprehensive analysis of the Linux kernel network stack to support decomposing more of it with our proposed redesign. Second, we need to investigate techniques for building and optimizing the TNA dependency graph, and to generate and deploy code based on it. Third, we need to come up with a model that can ensure correctness and consistency in the data plane as we insert custom processing. Finally, we will explore debugging mechanisms considering the new network stack design.

Acknowledgments

We thank Jennifer Rexford and the HotNets reviewers for their helpful feedback on this work. This work has been supported in part by NSF as part of CAREER award no. 1652698 and the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001.

References

[1] Marcelo Abranches and Eric Keller. 2020. A Userspace Transport Stack Doesn't Have to Mean Losing Linux Processing. In *IEEE NFV-SDN*. IEEE.

[2] Marcelo Abranches, Oliver Michel, Eric Keller, and Stefan Schmid. 2021. Efficient Network Monitoring Applications in the Kernel with eBPF and XDP. In *IEEE NFV-SDN*. IEEE.

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

[4] bridge 2012. Linux, bridge(8) — Linux manual page. https://man7.org/linux/man-pages/man8/bridge.8.html. (2012). Retrieved June 10, 2022.

[5] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding host network stack overheads. In *ACM SIGCOMM*.

[6] Cloudflare. 2017. Broken packets: IP fragmentation is flawed. https://blog.cloudflare.com/ip-fragmentation-is-broken/. (2017). Retrieved June 10, 2022.

[7] DPDK Project. 2022. Data Plane Development Kit. (2022). Retrieved June 13, 2022, from https://www.dpdk.org.

[8] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A Fast and Reliable Software Network Load Balancer. In *USENIX NSDI*.

[9] Open Networking Foundation. 2022. Aether. (2022). Retrieved June 16, 2022, from https://opennetworking.org/aether.

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

[11] FRR Project. 2022. FRRouting Project. (2022). Retrieved June 14, 2022, from https://frrouting.org.

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

[13] N. Hutchinson and L. Peterson. 1988. Design of the X-Kernel. In *ACM SIGCOMM*. 11. https://doi.org/10.1145/52324.52332

[14] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *USENIX NSDI*.

[15] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. 2000. The Click Modular Router. *Transactions on Computer Systems* 18, 3 (aug 2000), 35. https://doi.org/10.1145/354871.354874

[16] Teemu Koponen, Keith Amidon, Peter Balland, Martin Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan Jackson, Andrew Lambeth, Romain Lenglet, Shih-Hao Li, Amar Padmanabhan, Justin Pettit, Ben Pfaff, Rajiv Ramanathan, Scott Shenker, Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, and Ronghua Zhang. 2014. Network Virtualization in Multi-tenant Datacenters. In *USENIX NSDI*.

[17] Sebastiano Miano, Fulvio Risso, Mauricio Vásquez Bernal, Matteo Bertrone, and Yunsong Lu. 2021. A Framework for eBPF-Based Network Functions in an Era of Microservices. *IEEE TNSM* 18, 1 (2021).

[18] Jaehyun Nam, Seungsoo Lee, Hyunmin Seo, Phil Porras, Vinod Yegneswaran, and Seungwon Shin. 2020. BASTION: A Security Enforcement Network Stack for Container Networks. In *USENIX ATC*.

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

[20] Netronome Systems Inc. 2020. Netronome NFP-4000 Flow Processor Product Brief. (2020). Retrieved June 17, 2021 from https://www.netronome.com/media/documents/PB_NFP-4000-7-20.pdf.

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

[22] Cisco Press. 2018. BGP Fundamentals. https://www.ciscopress.com/articles/article.asp?p=2756480. (2018). Retrieved June 10, 2022.

[23] Luigi Rizzo. 2012. netmap: A Novel Framework for Fast Packet I/O. In *USENIX ATC*.

[24] Nikita Shirokov and Ranjeeth Dasineni. 2018. Open-sourcing Katran, a scalable network load balancer. (2018). Retrieved June 13, 2022, from https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer.

[25] strongSwan Project. 2022. strongSwan: the OpenSource IPsec-based VPN Solution. (2022). Retrieved June 16, 2022, from https://www.strongswan.org.

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

[27] weaveworks. 2022. Weave Net. (2022). Retrieved June 16, 2022, from https://www.weave.works/oss/net.

[28] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. 2021. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *ACM SOSP*. 17. https://doi.org/10.1145/3477132.3483569