Automatic Kernel Offload Using BPF

Farbod Shahinfar
Politecnico di Milano

Sebastiano Miano
Queen Mary University of London

Giuseppe Siracusano
NEC Laboratories Europe

Roberto Bifulco
NEC Laboratories Europe

Aurojit Panda
New York University

Gianni Antichi
Politecnico di Milano &
Queen Mary University of London

Abstract

BPF support in Linux has made kernel extensions easier. Recent efforts have shown that using BPF to offload portions of server applications, e.g., memcached and service proxies, can improve application performance and efficiency. However, thus far, the community has not looked at the question of what parts of an application should be offloaded? This paper first shows that blindly offloading application functionality to the kernel is neither beneficial nor desirable, and care must be taken when deciding what to offload. Furthermore, when deciding what to offload, developers must consider not just the application, but also the workload being handled, and the kernel being targeted, Therefore, we advocate automating this decision process in a compiler, that can analyze application code, and produce two executables, a kernel offload and a userspace program, that jointly implement the application's functionality. This paper discusses the challenges that must be addressed to build such a compiler, and why they can be feasibly addressed.

CCS Concepts

• **Networks** → Programming interfaces; **Data center networks**; *Cloud computing*; • **Software and its engineering** → **Source code generation**.

ACM Reference Format:
Farbod Shahinfar, Sebastiano Miano, Giuseppe Siracusano, Roberto Bifulco, Aurojit Panda, and Gianni Antichi. 2023. Automatic Kernel Offload Using BPF. In *Workshop on Hot Topics in Operating Systems (HOTOS '23), June 22–24, 2023, Providence, RI, USA*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3593856.3595888

ACM ISBN 979-8-4007-0195-5/23/06…$15.00
https://doi.org/10.1145/3593856.3595888

1 Introduction

BPF has made it easier to add functions to the Linux kernel: administrators can now load and execute sandboxed BPF programs that have limited access, without needing to recompile the kernel or trust a kernel module [3]. This new capability has attracted the interest of many researchers and systems builders, who have leveraged BPF to accelerate applications such as key-value stores [4], firewalls [12], service proxies [19] and virtual switches [26].

In this paper, we start from the observation that one needs to be careful when choosing what parts of an application to offload to the kernel. This is because, this choice affects both feasibility (can the offload run in the kernel) and performance (does the application run faster or more efficiently). Several prior works have focussed on the first aspect only by working around the kernel verifier limitations [4, 12, 13, 18], which must check the input BPF code for safety and can potentially make some code infeasible to offload. Our focus is not on this, since the community is actively working on removing its current limitations [9, 10].[1] We focus on both avoiding code that can never be feasibly offloaded, or if offloaded is unlikely to improve performance. Determining what offloads can improve performance turns out to be subtle and an important point to consider (§2).

Deciding what to offload requires significant programmer effort, since one must carefully consider code complexity, state dependencies, and what external libraries and system calls are used by different parts of the program. Therefore, we think that this analysis must be automated if we want BPF offload to truly benefit programmers.

In this paper, we suggest that a new compiler might be the right approach to *automatically offload program logic*. The core challenge lies in developing analysis that the compiler can use to identify code that should be offloaded (§2), and this is the main focus of our paper. We complement our discussion with an initial design of an end-to-end compiler that given the source code of a program, splits it into two executables, a userspace and an offload logic that has to be automatically attached to the most approapriate hook in the

[1]However, we believe that offloading some application logic will remain infeasible despite any planned or hypothetical improvements to the verifier.

kernel (§3). We conclude the paper with a discussion of the challenges associated to our design (§4).

2 To Offload or Not To Offload?

We start by looking at when offloading application logic to the kernel is beneficial, and when it might not be beneficial. The analysis we present here both motivates our proposal, and guides our proposed approach to automatically decide what to offload.

Much of the prior work [12, 13, 19, 26] has focused on offloading self-contained application logic that can be entirely executed in the kernel, and does not share state with any other parts of the application that might be running in userspace (in many cases the state requirements are trivially satisfied by having no userspace component). Other prior work [4], has used the kernel as a cache by memoizing the result of past computations, and using these memoized results to respond to requests when possible. In both of these cases, the performance benefits of offloading are because they allow the application to avoid a kernel-to-userpace transition.

Given this analysis of prior work, one might believe that the only code that should be offloaded is code that either processes requests entirely within the kernel, or acts as a fast path (such as a cache) for a significant fraction of requests. However, this is not the case, offloading can help in other circumstances too, including: (a) When it can summarize and thus reduce the amount of data that is returned by a system call (e.g., the `read` syscall), because in nearly all cases (with a few recent exceptions, e.g., io_uring with registered buffers, where no copies are performed), summarization reduces the amount of data the kenrel must copy from its buffers to application buffers; (b) By combining return values across multiple system calls, and thus batching system calls [8, 24] to reduce the number of kernel-to-userspace transitions; and (c) By offloading logic that can sometimes be executed entirely in the kernel (e.g., error code generation logic for web servers) thus avoiding a subset of kernel-to-userspace transitions. We note that our list is not exhaustive, it merely serves to illustrate that offloading can be beneficial even when the offloaded logic is not *self-contained*, and processing is split between the kernel and userspace.

However, care must be taken when offloading application logic that is not self-contained. We believe that there are at least four core concerns that must be considered when selecting what application logic is offloaded:

Feasibility. Not all application logic can be feasibly offloaded to the kernel. The BPF verifier used by the kernel poses one feasibility barrier, since it limits what logic can be loaded into the kernel. While the verifier has been improving steadily, allowing a wider variety of logic to be offloaded, we believe there will always be logic that cannot be offloaded. For example, it is unclear if the kernel will ever permit offloaded code to perform blocking operations (e.g., file reads or opens), since the overheads for blocking and then unblocking the thread negate any benefits from avoiding kernel-to-userspace crossing. We hasten to note this is different from syscall batching, which combines results from multiple syscalls all of which are issued by the userspace program. Regardless, it is easy to see that when choosing application logic to offload, programmers must consider feasibility.

Shared state. When looking beyond self-contained application logic, we must consider cases where the kernel and userspace logic share state. Unfortunately, the standard technique to sharing state between kernel-and-userspace requires creating BPF maps in the kernel, which can then be accessed using read and write syscalls, or by using `mmap` to map it into userspace [16]. Both add significant overheads to access, and thus programmers must consider the frequency of shared state access when deciding on what application logic to offload to the kernel.

Synchronization. Synchronization might be necessary when splitting application state and logic between the kernel and userspace. Of course, synchronization across isolation domains (i.e., the kernel and userspace) is likely to be more expensive than synchronization within an application's threads (for instance, any synchronization primitive requires state access, which as we explained above carries significant overhead in this context). Therefore, programmers must also consider synchronization frequency when choosing what to offload.

Benefits. Finally, and perhaps most importantly, programmers need to evaluate whether offloading to the kernel is beneficial to their program. The benefits of offloading specific application logic depend not only on the logic and how it interacts with the rest of the application, but also on the workload, which determines how frequently it is executed. At the same time, offloading code which has no benefit into the kernel comes at a cost, and affects system complexity and performance.

Our message is not that the factors listed above are exhaustive, in fact we believe it is likely that other factors that we missed also have a large impact on decisions about what application logic should be offloaded. Instead, what we aim to convey is that programmers must analyze several factors when deciding what application logic should be offloaded, and the observation that few if any tools are available to help the programmer with this analysis: tools that analyze shared state accesses, or synchronization between code paths are rare if not non-existent (we do not know of any), and no tools have targeted the type of feasibility analysis required for this usecase. Therefore, we believe that this analysis poses a significant programmer's burden, and prevents the wider adoption and use of kernel offloads.

Thus, we argue that we should build tools to automate this analysis and eliminate this programmer's burden. Specifically, we envision an approach where a compiler splits an input program into two parts, one that runs in userspace and another that is offloaded to the kernel, that jointly implement the input program's functionality. The key challenge in building such a compiler lies in how we analyze programs and automatically make determinations on the factors we listed above.

2.1 Can we automate offload selection?

Is it feasible for us to analyze programs and determine what to offload? Below we briefly outline approaches for analyzing each of the four factors we highlight above, that we think are implementable today and suffice for our purposes:

Feasibility. Our feasibility concerns come from two sources: does the selected code make calls that no kernel would offload, and can the selected code pass the current kernel verifier? Checking for the former requires using syntactic rules, similar to what any linter does, and is thus within reach. Prior work [28] uses the kernel verifier in userspace to check this, and has also shown that super-optimization or other techniques can often be used to optimize the code (in a semantic preserving manner) to pass the verifier.

Shared State and Synchronization. Existing approaches to program slicing [1, 23, 27] can already split a program into independent slices with no data or control flow dependencies. These approaches suffice for producing independent components that do not need shared state, and in most cases do not need to synchronize with each other. However, program slicing might be too extreme for our use: we think in many cases it might be necessary and even desirable, to offload application logic that shares state with the userspace program, and we merely need to ensure that userspace access is infrequent. Similarly, it might be desirable to not entirely disallow synchronization, but merely ensure that it is infrequent. We believe that this can be done by using an approach similar to profile guided optimization [7], and using profile information (gathered from executing the program) to inform the program slicing algorithm.

Benefits. Finally, evaluating benefits requires modeling and comparing the performance of different choices of what application logic to offload. Prior work [5, 6] has shown the feasibility of using models to predict relative performance, and in combination with profiling can allow us to empirically check benefits and determine whether and what application logic should be offloaded.

Thus, we believe that we can automate the determination of what application logic to offload, however we are still working on actually validating our approach. In the next section, we discuss how this analysis fits into a compiler to implement the end-to-end workflow we target.

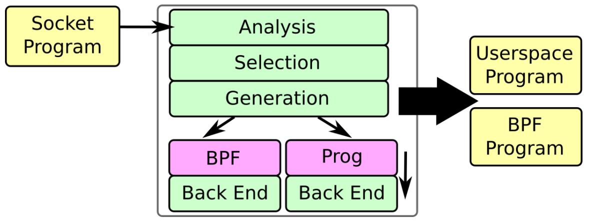

Figure 1: A high level overview of steps for automatically generating BPF offload programs. The source code is analysed for extracting the application logic. The paths which benefit from offloading are selected and transformed into a BPF program.

3 How to Automatically Offload?

In this section we present an overview of our envisioned compiler (Figure 1) and end-to-end workflow. Given the original application's code, our solution first extracts the program's control flow graph (CFG) (§3.1). Then, it uses the analysis function we proposed in the previous section to select what code should be offloaded, and selects the appropriate hooks where the offloaded code should be installed (§3.2). Finally, it creates two executables: one for the BPF offload and one for the userspace application (§3.3).

3.1 Understanding the source code

In order to automatically understand the logic of an application, we make three assumptions about its structure: (1) The application uses POSIX API: this is useful as we can rely on standard function calls to understand program behavior (Table 1 provides a list of functions that we consider and the information they convey); (2) The source code contains event-loops which handle the requests: this helps to quickly find the logic that can be potentially offloaded; (3) The CFG of the application is known during the compilation (i.e., there are no indirect calls or jumps): this is important as without that we cannot split the input program into two executables at compile time. Such type of requirements are usually a common practice and past works have assumed a known code structure as well [15, 21].

The idea is to first scan the code in search for an event-loop (*second assumption*) which is started with the function call `poll` (*first assumption*). After that, the first time we find the call to the `recv` function indicates the arrival of a request and root of our CFG. We deduced the end of the request processing in the following two cases: (1) When finding one or more invocation of the `send` function call: in this case the program ends with a reply action; (2) If there is no invocation of the `send` function call until the next call to the `poll` function. Our reasoning is that when translating this path to BPF, the packet resulted in the execution of this path should be dropped anyway until further packets arrive.

Table 1: List of landmark functions

Functions	Indicator	Information
`bind`	Which request are related to this service	Server address
`poll`, `select`, `epoll`	Beginning of the event-loop	–
`recv`, `read`, `recvfrom`, `recvmsg`	Arrival of a request	Receive buffer
`send`, `write`, `sendto`, `sendmsg`	Replying to a request	Send buffer

It is worth noting that during the analysis, it is possible to encounter multiple calls to `recv`. In this case, the calls are grouped together and considered as one, if they are on the same socket. However, reading from different sockets would be interpreted as IO operations on the execution path.[2] Similarly, sending data on multiple sockets is treated as multiple IO operations (e.g., RPC or database queries).

3.2 Selecting The Right BPF Hook

After selecting the paths that shall be offloaded from the application CFG, the next step is to pick the right BPF hook(s) in the kernel. This is an important step as each hook has access to different parts of a packet and different helper functions: in Table 2, we present a list of currently available hooks alongside examples on when they can be useful.

Naively, we might be inclined to attach the BPF program to the lowest possible hook: for packet reception this is XDP, which is located in the NIC driver just after the interrupt processing and before any operations performed by the network stack. Indeed, by doing so it would be potentially possible to save computation and improve performance by quickly processing the incoming packet and enable an *early-exit* path. Unfortunately, this is not always possible. Indeed, when offloading a socket program to BPF, the target hook must be chosen based on both the transport protocol and semantic of the program itself.

Let's take as an example a TCP-based application which gets accelerated by offloading part of its logic into the kernel. Here, if we would attach the offload to XDP or TC, in case of early-exit (i.e., the offload replies directly to the source), the kernel would not have visibility on this action and could wrongly treat the packet handled by the offload as lost, triggering a retransmission. To avoid that, it is important to attach the BPF program to the SK_SKB hook which is located after the kernel has provided its TCP processing.

In contrast, for UDP-based application the situation is different. They are not as restricted as the TCP ones and here both the XDP and TC hooks provides enough flexibility to attach the BPF offload.

[2]System calls and IO operations are not supported in BPF and paths having IO operations will be ignored.

3.3 Generating the Executables

The last and final step requires us to convert the selected application CFG paths into two executables: one BPF for the offload and one for userspace.

Fortunately, BPF compilers are available for most languages, and we plan to reuse them for our work. The only thing we need to do before invoking these compilers is to replace calls that send or receive data (e.g., `recv` and `send`) with kernel alternatives, and to handle state. The former is a simple syntactic change.

State requires a little more efforts, since any global or shared variables must be implemented using BPF maps. Finally, the compiler has to add the bound checking instructurs before accessing the packet data and bound any unbounded loop to make the verifier happy. In case the result is not found in the bounded offloaded loop, then the application shall fall back to the userspace logic.

We also need to change the userspace executable to remove offloaded codepaths, to replace calls into this codepath with appropriate calls, and to change how shared variables are accessed. We plan to use `mmaped` access for the later.

4 Discussion

The analysis step described in §2 is the core challenge that needs to be addressed to achieve our vision of building a compiler that automates the use of eBPF offloads. However, once this challenge has been addressed, and a compiler has been built, new challenges and questions emerge. We discuss these below.

External Libraries. Thus far in our description we focused on applications whose code is self-contained. But most applications use a variety of libraries. While in some cases, we can consider library code during our analysis (i.e., perform the equivalent of link-time optimization [25]), and include portions of it in our offload binary, this requires that we either have access to the library's code or to an intermediate form (e.g., Rust `rlibs` or `dylibs`). However, source code is not available for all libraries, and most libraries do not ship in a form that is amenable to analysis or offloading. A naive approach to deal with this problem is to never offload any application logic that uses an external library, but this is limiting. New approaches, either technical (e.g., ways to link in library code in BPF) or cultural (e.g., changes that make analyzable libraries a common occurrence), will be required to address this problem.

Handling Workload Changes. Many of our initial suggestions for how to identify application logic to offload are profile driven. We do not know of alternate approaches to analyze some of the criterion (e.g., frequency of shared state access, or benefit) that we think are important when deciding

Table 2: Description of different BPF hook. It shows the direction of data path (I: ingress, E: egress) and the information a hook can access.

Name	I/E	Input	Conditions	Example Application
XDP	I	Packet	Processes packets before network stack. It is suitable for offloading stateless protocols like UDP. But dropping or modifying TCP packets can corrupt the connection state.	Load-Balancing, Active Queue Management, Access Control
TC	I/E	SKB	Can classify packets and redirect them to another destination.	Packet Classification and routing.
SK_LOOKUP	I	SKB	Runs when network stack is selecting the target socket. It can redirect connections to another socket.	Binding a service (one socket) to multiple addresses and ports.
SK_SKB	I	SKB	Runs on TCP stack receive data event. It can modify packets and respond to them	Accelerating NGINX.
SK_MSG	E	MSG	Runs before the message enters the network stack. It can modify and redirect the packet to another socket.	Offloading TLS to the kernel. Accelerating proxy servers.

what to offload. However, profiles depend on workloads, and workloads change at both small (e.g., over hours due to the diurnal effect) and large (e.g., over months due to changing needs) time intervals. Thus, our choice of what to offload might be suboptimal over time, and lead to lower performance gains, or even worse performance than no offloads. Therefore, we believe that in addition to the compiler that has been the focus of this paper, an approach such as ours also needs a runtime that detects workload changes, and reruns our compiler when necessary. Recent work [15] has proposed related approaches to run time optimization that demonstrate the feasibility of such an approach.

Beyond Network Applications. Our description in §3 focused on offloading network functionality from applications. However, a similar approach can be adopted to offloading other functionality, e.g., storage based functionality.

Consider, for instance, how we might automate the use of XRP [29], a recently proposed framework that uses BPF hooks in NVMe drivers to accelerate an application's storage functionality. The core change required is to develop a new approach to find entry points in the code for our analysis (§3.1), since we can longer depend on tracing code from an event loop. We can take a few approaches in this case: we can require programmer's to annotate functions where the compiler should begin its analysis, or we can require code to be written in a DSL that makes it easier for use to identify entry-points. Of course, the approach chosen impacts both applicability (i.e., what programs can we target), and efficacy (i.e., how much can the compiler offload).

Impact on kernel evolution. Finally, our focus thus far has been on how to make it easier for applications to offload logic to the kernel, allowing a wider variety of applications to make use of these capabilities. We believe that this should also influence how the kernel evolves in the future. Some of this evolution is trivial, perhaps a change in how offloads are used might suggest new kernel functions that should be exported so they can be used from BPF. But perhaps this can also lead to a deeper discussion of additional kernel functionality or modules that might benefit applications: we have already seen this happen before, when wider use of TLS led to the kernel adopting kTLS [11], allowing some TLS processing to move from userspace libraries into the kernel.

5 Related Work

BPF-based approaches for improving application efficiency. Many papers from the research community or industry effort have leveraged BPF for improving application performance [4, 12, 14, 15, 17, 18, 22, 26, 28, 29]. Studies such as BMC [4], bpf-iptables [12], and SPRIGHT [18] aim to improve the performance of applications such as Memcached, iptables, and Kubernetes by manually offloading self-contained application logic that can be executed solely in the kernel, without sharing state with other parts of the application running in userspace. Building on these findings, our work goes a step further by proposing that even when application logic is not fully self-contained, offloading part of its functionality can still yield performance benefits. However, care must be taken to avoid offloading code that does not provide benefits, as this can increase system complexity and negatively impact performance.

Static and dynamic optimization of BPF programs. Several works have explored the possibility to either statically or dynamically optimize BPF programs. K2 [28] is a program-synthesis-based compiler that automatically optimizes BPF bytecode with formal correctness. It synthesizes programs that are formally shown to be equivalent to the original program and uses a cost function to limit and guide its search process. Morpheus [15] describes and implements a system that instruments packet-processing programs to gather statistics at runtime, and supports both eBPF and DPDK backends. It detects packet-level dynamic to apply aggressive optimization depending on the specific workloads. Unlike these works, we propose to analyse socket programs in order to generate BPF programs. This is orthogonal to their system. Both Morpheus and K2 can be used to further optimize the automatically generated BPF offload.

Simplifying BPF programming. Writing BPF programs is known to be difficult and challenging [4, 28]. To tackle these challenges, some solutions have been proposed. For example, bpftrace [2] provides a scripting language that compiles to BPF, making it easier to write BPF programs for tracing and observability. Others [20] propose using the flexibility of userspace to write BPF programs without worrying about the verifier, relying on a compiler to split the code into BPF and userspace programs. Our work is complementary to these solutions, as we aim to address a different problem, how to automatically identify and generate BPF offloads and userspace programs.

6 Join the Revolution

BPF and other new Linux interfaces are usually added to improve security, enable new application functionality, or improve the performance of existing applications. However, using these new interfaces poses a signficant challenge for application developers: they must learn and understand the tradeoffs these new interfaces offer, understand the maintainence complexities that might come from adopting them, and potentially loose compatability with other UNIX-like systems. Consequently, many of these features are only ever used by hyper-scalers and companies that can afford an army of experts, or by research groups where graduate students and postdocs use them in projects which are generally not maintained after publication. In this paper we proposed a path that allows one new interface, BPF, to be more widely adopted, by automating away much of the complexity that programmers have to deal with today. We think the systems research and the Linux development communities ought to consider ours and other approaches to automating interface adoption, since it is the only chance we have to democratize who benefits from these improvements. Viva la revolución.

Acknowledgements. We thank the HotOS reviewers for their useful feedbacks. This work was funded in part by the National Science Foundation grant 2137220, the UK EPSRC project EP/T007206/1, and by the European Union's Italian National Recovery and Resilience Plan (NRRP) of NextGenerationEU, partnership on "Telecommunications of the Future" (PE00000001, "RESTART").

References

[1] David W Binkley and Keith Brian Gallagher. 1996. Program slicing. *Advances in computers* 43 (1996), 1–50.

[2] bpftrace 2022. bpftrace: high-level tracing language for Linux enhanced Berkeley Packet Filter. https://github.com/iovisor/bpftrace.

[3] ebpf 2022. *eBPF.* https://ebpf.io/

[4] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. 2021. BMC: Accelerating Memcached using Safe In-kernel Caching and Pre-stack Processing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21).* 487–501.

[5] Rishabh Iyer, Katerina Argyraki, and George Candea. 2022. Performance Interfaces for Network Functions. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22).* USENIX, 567–584.

[6] Rishabh Iyer, Luis Pedrosa, Arseniy Zaostrovnykh, Solal Pirelli, Katerina Argyraki, and George Candea. 2019. Performance Contracts for Software Network Functions. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation (NSDI'19).* USENIX, 517–530.

[7] Erik Johansson and Sven-Olof Nyström. 2000. Profile-Guided Optimization across Process Boundaries. *SIGPLAN Not.* 35, 7 (jan 2000), 23–31.

[8] Ake Koomsin and Yasushi Shinjo. 2015. Running Application Specific Kernel Code by a Just-in-Time Compiler. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems (PLOS '15).* ACM, 15–20.

[9] Joanne Koong. 2021. *A different approach to BPF loops.* https://lwn.net/ml/bpf/20211123183409.3599979-1-joannekoong@fb.com/

[10] Joanne Koong. 2022. *BPF: Dynamic pointers.* https://lwn.net/Articles/895885/

[11] ktls 2022. *Kernel TLSD Offload.* https://docs.kernel.org/networking/tls-offload.html

[12] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Mauricio Vásquez Bernal, Yunsong Lu, and Jianwen Pi. 2019. Securing Linux with a Faster and Scalable Iptables. *SIGCOMM Comput. Commun. Rev.* 49, 3 (nov 2019), 2–17.

[13] Sebastiano Miano, Roberto Doriguzzi-Corin, Fulvio Risso, Domenico Siracusa, and Raffaele Sommese. 2019. Introducing SmartNICs in Server-Based Data Plane Processing: The DDoS Mitigation Use Case. *IEEE Access* 7 (2019), 107161–107170.

[14] 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 Transactions on Network and Service Management* 18, 1 (2021), 133–151.

[15] Sebastiano Miano, Alireza Sanaee, Fulvio Risso, Gábor Rétvári, and Gianni Antichi. 2022. Domain Specific Run Time Optimization for Software Data Planes. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22).* ACM, 1148–1164.

[16] Andrii Nakryiko. 2019. *Add support for memory-mapping BPF array maps.* https://lwn.net/Articles/805043/

[17] Tomasz Osiński, Jan Palimąka, Mateusz Kossakowski, Frédéric Dang Tran, El-Fadel Bonfoh, and Halina Tarasiuk. 2022. A Novel Programmable Software Datapath for Software-Defined Networking. In *Proceedings of the 18th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '22).* ACM, 245–260.

[18] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and K. K. Ramakrishnan. 2022. SPRIGHT: Extracting the Server from Serverless Computing! High-Performance EBPF-Based Event-Driven, Shared-Memory Processing. In *Proceedings of the ACM SIGCOMM 2022 Conference (SIGCOMM '22).* ACM, 780–794.

[19] Liz Rice. 2021. *Cilium: How eBPF Streamlines the Service Mesh.* https://thenewstack.io/how-ebpf-streamlines-the-service-mesh/

[20] Farbod Shahinfar, Sebastiano Miano, Alireza Sanaee, Giuseppe Siracusano, Roberto Bifulco, and Gianni Antichi. 2021. The Case for

Network Functions Decomposition. In *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '21)*. ACM, 475–476.

[21] Justine Sherry, Peter Xiang Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocco, Maziar Manesh, João Martins, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. 2015. Rollback-Recovery for Middleboxes. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. ACM, 227–240.

[22] Nikita Shirokov and Ranjeeth Dasineni. 2018. *Katran, a scalable network load balancer.* https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/

[23] Josep Silva. 2012. A Vocabulary of Program Slicing-Based Techniques. *ACM Comput. Surv.* 44, 3, Article 12 (jun 2012).

[24] Livio Soares and Michael Stumm. 2010. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX, 33–46.

[25] Amitabh Srivastava and David W. Wall. 1994. Link-Time Optimization of Address Calculation on a 64-Bit Architecture. *SIGPLAN Not.* 29, 6 (jun 1994), 49–60.

[26] William Tu, Yi-Hung Wei, Gianni Antichi, and Ben Pfaff. 2021. Revisiting the Open VSwitch Dataplane Ten Years Later. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM '21)*. ACM, 245–257.

[27] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. 2005. A Brief Survey of Program Slicing. *SIGSOFT Softw. Eng. Notes* 30, 2 (mar 2005), 1–36.

[28] Qiongwen Xu, Michael D. Wong, Tanvi Wagle, Srinivas Narayana, and Anirudh Sivaraman. 2021. Synthesizing Safe and Efficient Kernel Extensions for Packet Processing. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM '21)*. ACM, 50–64.

[29] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. 2022. XRP: In-Kernel Storage Functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX, 375–393.