Journal of Network and Computer Applications 222 (2024) 103805

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

Research paper

Network shortcut in data plane of service mesh with eBPF

Wanqi Yang [a], Pengfei Chen [a,*], Guangba Yu [a], Haibin Zhang [b], Huxing Zhang [b]

[a] School of Computer Science and Engineering, Sun Yat-Sen University, Guangzhou, China
[b] Alibaba Group, China

ARTICLE INFO

Keywords:
Service mesh
eBPF
Network optimization
Istio
Socket redirect

ABSTRACT

In recent years, the adoption of the service mesh as a dedicated infrastructure layer to support cloud-native systems has gained significant popularity. Service meshes involve the incorporation of proxies to handle communication between microservices, thereby speeding up the development and deployment of microservice applications. However, the use of service meshes also increases the request latency because they elongate the packet transmission between services. After investigating the transmission path of packets in a representative service mesh Istio, we observed that the service mesh dedicates approximately 25% of its time to packet transmission in the Linux kernel network stack. To shorten this process, we propose a non-intrusive solution that enables packets to bypass the kernel network stack through the implementation of socket redirection and tc (traffic control) redirection with eBPF (extended Berkeley Packet Filter). We also conduct comprehensive experiments on the widely-used Istio. The evaluation results show that our approach can significantly reduce the request latency by up to 21%. Furthermore, our approach decreases CPU usage by 1.73% and reduces memory consumption by approximately 0.98% when compared to the original service mesh implementation.

1. Introduction

The microservice architecture decomposes large, self-contained monolithic applications into smaller, loosely-coupled services, enhancing flexibility and scalability, as each service can be developed, tested, and deployed independently (Yu et al., 2019; Srirama et al., 2020; Wan et al., 2018; Cinque et al., 2022). In microservice systems, communication between services is facilitated by software development kits (SDKs), which ensure consistent service invocation across the microservice network. However, executing custom-built SDKs in microservice can be challenging, as it requires a significant development effort and sophisticated control, and may result in inconsistencies due to human errors.

The service mesh is introduced to decouple the dependencies between SDKs for faster deployment and development of microservices. A service mesh serves as a dedicated infrastructure layer, deploying and managing network proxies in conjunction with each microservice instance (e.g., Fig. 1(b)). These proxies intercept and control network communication of microservices, without requiring any modifications to their source codes. According to the 2022 Annual APIs and Integration Report (AG, 2022), over 47% of organizations have adopted service meshes to manage their microservice systems.

Although the introduced proxies in a service mesh essentially eliminate the maintenance overhead of SDKs, they result in increased latency due to longer service-to-service communication. As shown in Fig. 1, the communication between service instances S_1 and S_2 becomes longer in a service mesh compared to that in a traditional microservice architecture. In the traditional cases, S_1 and S_2 communicate directly with each other, as the service and communication SDK are packaged within the same container. In a service mesh, unlike direct communication, S_1 first sends the request to proxy P_1, which then forwards it to proxy P_2 and ultimately to S_2. The response transmission follows a similar path in reverse. This procedure demonstrates that service communication in a service mesh involves multiple interactions with proxies, leading to increased latency.

To better understand the impact of service mesh on request latency, we conduct an experiment with the microservice application Bookinfo (2022) and the typical service mesh Istio (2022), whose details will be shown in Section 3.3. Our results show that Istio introduces an additional 14 ms (ms) request latency overhead compared to the 16 ms latency experienced in the microservice architecture. The additional latency comes from the fact that requests and responses have to pass through proxies as well as the underlying network stack multiple times. In addition, service meshes typically manage the packet forwarding and filtering rules based on iptables (Purdy, 2004), which has an inefficient computational complexity of $O(n)$. Service meshes have to spend more time matching rules in a large iptable in the network stack, which is important for alleviating the increased request latency.

* Correspondence to: School of Computer Science and Engineering, Sun Yat-Sen University, Guangzhou, Guangdong 510006, China.
E-mail address: chenpf7@mail.sysu.edu.cn (P. Chen).

https://doi.org/10.1016/j.jnca.2023.103805
Received 5 July 2023; Received in revised form 1 November 2023; Accepted 28 November 2023
Available online 30 November 2023

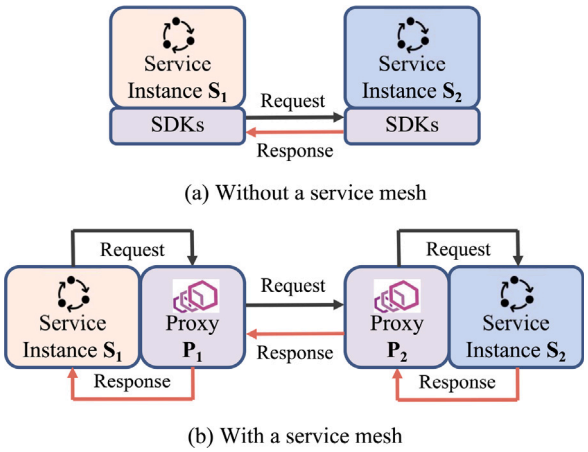


Fig. 1. The communication between service instance S_1 and service instance S_2 in a microservice system with and without service mesh deployed.

Existing approaches (Merbridge, 2022; Xu Yizhou, 2021; Cilium, 2020) fail to effectively mitigate the adverse effects of service meshes. The performance improvement of Merbridge (2022) and Xu Yizhou (2021) is limited because they are designed to optimize packet transmission within a single node and are unable to extend to inter-node transmission. Cilium (2020) can speed up the inter-node packet transmission by implementing its own service mesh from scratch. However, Cilium's optimization is dependent on its own Kubernetes container network interface (CNI) cilium-cni. Therefore, the Kubernetes cluster deployed on other popular CNIs (e.g., Calico (2023)) has to be redeployed. Redeploying microservices can result in the unavailability of business services temporarily, which is unacceptable in production environments due to huge loss in revenue.

To overcome the above limitations, we propose a non-intrusive framework to reduce the impact of service mesh on request latency. We first collect latency distribution of packet transmissions inside a service mesh using eBPF (extended Berkeley Packet Filter) (Corbet, 2014) (Section 3.3). We observe that the request latency is primarily distributed among three parts: Istio proxies, kernel network stack and application itself, which account for 25.83%, 25.4% and 48.77%, respectively. Since the delay consumed in the proxies and applications is intrinsic in service meshes, it cannot be eliminated in a non-intrusive manner. Thus, we attempt to shorten the packet transmission path in the kernel network stack to alleviate the increased latency incurred by service meshes. The core idea is to utilize eBPF to enable socket redirection and tc (traffic control) redirection, allowing packets to bypass the underlying network stack.

Specifically, we propose two optimization strategies, namely intra-pod optimization and inter-pod optimization. For intra-pod optimization, requests and responses are transmitted directly between sockets. For inter-pod optimization, requests and responses can also be redirected between sockets within a node, and be redirected between two NICs (Network Interface Card) for across-node communication.

Practical evaluations on two widely-used microservice benchmarks, Bookinfo (2022) and Google (2022), driven by a representative service mesh Istio (2022) demonstrate that our approach improves the request latency by up to 21% for Bookinfo and up to 10% for HipsterShop, mainly improving latency for the first 90% of requests. In addition, our proposed optimization can both reduce the overhead of CPU utilization and memory usage.

In summary, this paper makes the following contributions.

- We construct an eBPF monitor to obtain the latency distribution of service-to-service communication in a typical service mesh and analyze that service mesh has a noteworthy effect on request latency, where the kernel network stack accounts for about 25%.
- We shorten service-to-service communication in the service mesh with eBPF in a non-intrusive way to alleviate the increased latency incurred by service meshes. Especially, our approach supports the optimization on cross-node communication under large-scale clusters.
- We evaluate the performance of our approach on Bookinfo and HipsterShop deployed with Istio. We find that our approach can reduce the request latency by up to 21%, CPU utilization by 1.73%, and memory by 0.98%.

The following sections of this paper are organized as follows. In Section 2, we discuss some related work on eBPF and network shortcuts in Linux. Section 3 introduces the background of service meshes and eBPF as well as our motivation for doing the network shortcut in the service mesh. In Section 4, we show the design of our optimization. We evaluate the performance of our proposed approach and provide a discussion in Section 5. Finally, Section 6 concludes this paper.

2. Related work

eBPF allows user-defined programs to run in the Linux kernel. Thus, eBPF plays an important role in performance monitoring. Brondolin and Santambrogio (2020) proposes a black-box monitoring framework to help measure microservice runtime performance. bpftrace (2018) realizes eBPF monitors to collect kernel and user program runtime information, which improves the observability of Linux kernel significantly. Nam and Kim (2017) implement eBPF-based packet tracing on multiple Linux network interfaces. Lee et al. (2022) enhance packet tracing with eBPF to help latency measurement. Abranches et al. (2021) introduce a network monitoring approach that enables efficient high-level metric computation with eBPF and XDP. Miano et al. (2023) implement high-performance network measurement in eBPF, like sketch-based algorithms. Deepflow (Shen et al., 2023) proposes a distributed tracing framework for troubleshooting microservices with monitoring data provided by eBPF.

There also exists some optimization work on network functionality based on eBPF. InKeV (Ahmed et al., 2018) enables programmable distributed network virtualization for DCN. BPFabric (Jouet and Pezaros, 2017) enhances the packet processing and forwarding functionality of the data plane with the programmability of eBPF. SPRIGHT (Qi et al., 2022) uses eBPF to support shared memory processing and implements serverless computing. Baidya et al. (2018) implements real-time packet replication and forwarding based on eBPF. Facebook (2018) and Xhonneux et al. (2018) implement a load balancer and segment routing in the kernel with eBPF. Scholz et al. (2018) analyzes the impact of eBPF's XDP programs running in the NIC. Choe et al. (2020) defends against DoS attacks by XDP filtering. Miano et al. (2018) summarized the experience of constructing the complex network with eBPF. Electrode (Zhou et al., 2023) accelerates the performance of distributed protocols by implementing the election process with eBPF. MiddleNet (Qi et al., 2023) utilizes eBPF to achieve high-performance communication in L4/L7 function chains. TPC (Jadin et al., 2022) leverages eBPF to manage TCP connections and make TCP path-aware. BPF can also used to program packet filters or basic forwarding in P4 (Hauser et al., 2023). BCC (2022) and libbpf (2020) are proposed to help developers rapidly develop eBPF programs in high-level languages such as Golang and Python. In this paper, we develop eBPF programs with libbpf since libbpf consumes fewer system resources (e.g., CPU) than BCC.

Kernel bypassing is an important approach for network optimization, even in service meshes. Data Plane Development Kit (DPDK) (Intel, 2014) and NetMap (Rizzo, 2012) speed up Linux packet processing by bypassing the network stack and processing packets in user space in a polling mode. AF_XDP (Kuhr and Carôt, 2020) directly forwards packets to corresponding user programs. IO-TCP (Kim et al., 2023) splits the TCP stack and offloads disk I/O and TCP packet transfer to SmartNIC to accelerate content delivery. Intel utilizes eBPF to accelerate the container network of Xu Yizhou (2021). Merbridge (2022)

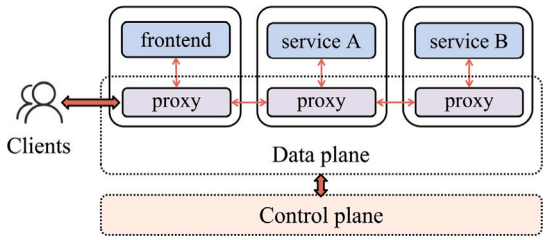


Fig. 2. A typical architecture of a service mesh.

accelerates packet transmission between socket pairs in a node by using eBPF instead of iptables in a service mesh. Cilium (2020) implements its own service mesh by deploying an eBPF proxy per node. Compared to the existing network optimization approaches in Istio, our approach can better handle optimization on the cross-node communication between services in a large-scale microservice system, requires no intrusive codes into applications and service meshes, and does not need any redeployment of the Kubernetes cluster.

3. Background and motivation

3.1. Service mesh

Nowadays, containers are adopted in microservice systems since they virtualize system resources in a lightweight way. Kubernetes (2022) helps manage containers and allows developers to automatically deploy and scale services (Bernstein, 2014), where a pod is the basic scheduling unit. One or more containers can be deployed in one pod, interacting with each other by the Loopback NIC (Network Interface Card). Flannel (2014) is one kind of the underlying network implementations of Kubernetes, which assigns each node a network subnet and transmits packets over the VxLan (Mahalingam et al., 2014).

Although microservice improves the scalability of applications, it introduces some new problems. In microservice systems, developers have to embed service invocation in source codes, for example, if there is a dependency on the service version, which makes application migration and upgrade difficult. Service meshes (Li et al., 2019) were proposed to handle service communication and implement reliable packet delivery in cloud-native applications. Service meshes are usually implemented with lightweight proxies, which are deployed together with service instances and transparent to applications. In a service mesh, services only focus on their own business, with the communication between services implemented by proxies.

Fig. 2 shows the architecture of a typical service mesh. The connection between proxies represents the invocation between services. A topology formed by proxies constitutes a service mesh, where the collection of these interconnected proxies is called the data plane. All the inbound and outbound traffic for each service instance must pass through its proxy first. Configured by the control plane, proxies finish the load balancing and forward traffic to the corresponding service instances. Istio is one of the most popular service meshes (Calcote and Butcher, 2019; Istio, 2022), where an envoy (Envoy, 2019) works as a high-performance proxy for all inbound and outbound traffic forwarding. In this paper, we choose Istio to perform our optimization for the network shortcut.

3.2. eBPF

BPF (Berkeley Packet Filter) (McCanne and Jacobson, 1993) was proposed to capture and filter packets that meet certain rules as soon as possible, avoiding packets copying from kernel space to user space in Linux and improving the performance of packet processing. tcpdump (tcpdump, 2022), a widely-used Linux tool, is implemented based on BPF. Extended BPF (eBPF) (Corbet, 2014; eBPF, 2022) is designed to allow developers to run user-defined programs in the kernel and extend the functionality of BPF. Developers first write an eBPF program and then compile it into bytecode with Clang and LLVM (Low Level Virtual Machine). After passing the security verification by the eBPF verifier, the bytecode is converted into the machine code by eBPF JIT (Just-in-Time) compiler and finally attached to somewhere in the kernel, which is called a hook. Each time a program executes at the attached hook, the eBPF program is triggered to run. eBPF is useful for packet filtering, kernel debugging, and more (Vieira et al., 2020).

3.2.1. eBPF program types

eBPF defines many types of programs as well as their mounted hooks in the kernel. In this paper, we mainly utilize *kprobe*, *sockops* and *sk_msg* and *tc* of the eBPF program types to realize our optimization. *kprobe* programs can be attached to the entry of kernel functions to capture fine-grained metrics in the kernel. *sockops* programs are triggered when socket connections are established, obtaining the connection information such as IP addresses and ports of the established sockets. *sk_msg* programs are attached to a map storing specific sockets and triggered to execute when those sockets send a file or a message. *tc* programs are attached to the inbound or outbound queue of a NIC, processing the packets passing through.

3.2.2. eBPF maps

eBPF Maps (eBPF, 2022) are in-kernel key–value data stores specifically designed for eBPF programs. eBPF Maps can be accessed by both eBPF programs and user programs, sharing data among them. eBPF Maps have multiple types such as *hash*, *array*, *sockhash* and so on. In this paper, we use the *hash* and *sockhash* of eBPF Maps. *hash* acts as a hash table. *sockhash* is a hash table with user-defined keys, storing sockets as values.

3.3. Motivation

To better understand the distribution of request latency in the service mesh, we apply eBPF to collect the occurrence timestamp of socket functions on sending and receiving data, without intrusiveness to the service mesh. eBPF *kprobe* programs are attached to the kernel function *sock_sendmsg* and *sock_recvmsg*. When the established sockets start to send or receive messages, eBPF programs are triggered to collect their occurrence timestamp. This allows us to calculate the time taken for packets to pass through Istio proxies, the kernel network stack and the applications.

We denote the receiving timestamp of socket(i) as T^i_{recv} and its sending timestamp as T^i_{send}. As shown in Fig. 3, eBPF *kprobe* programs monitor the established sockets for their receiving and sending timestamp. The solid lines indicate packet transmission processes. Blue lines indicate that packets are processed by proxies, black lines represent the time for packets to travel through the kernel network stack, and orange lines denote request processing time within the applications respectively. Table 1 shows the time overhead of these three processes and displays their distribution with 300 requests randomly issued in the Istio benchmark *Bookinfo* (Bookinfo, 2022).

From Table 1, we can observe that Bookinfo serves clients with an average request latency of 30.31 ms (ms) when it is managed by Istio, while that without Istio is 16.44 ms. In other words, Istio imposes an 84% latency overhead on Bookinfo. The additional latency comes from the additional time for packets to pass through the Istio proxies and the kernel network stack multiple times. Alleviating the increased request latency incurred by service meshes is imperative to provide better performance for developers.

In order to optimize service meshes, we go deep into the packet transmission path within the service mesh. As shown in Table 1, the request latency is primarily spent in three parts: the Istio proxies, the kernel network stack and the application itself, which account for

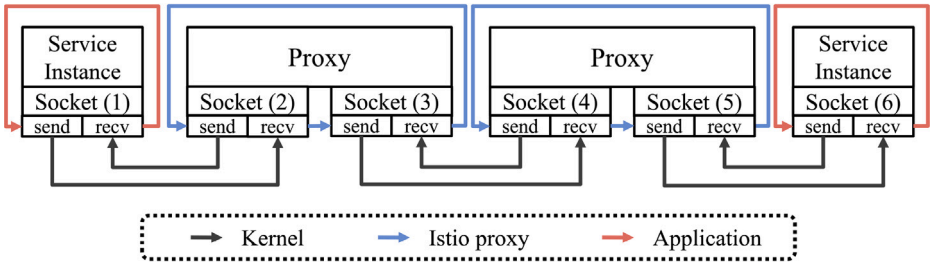
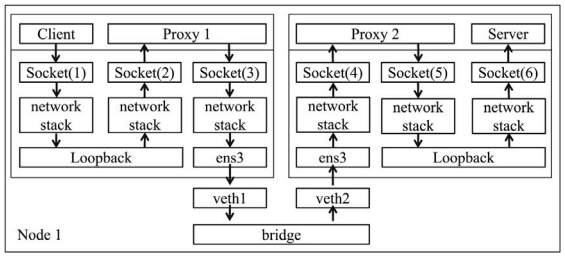


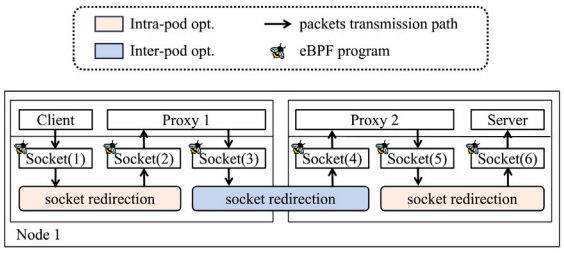
Fig. 3. The procedures of service-to-service communication in Istio.

Table 1
The distribution of average latency when deploying Bookinfo application with and without Istio.

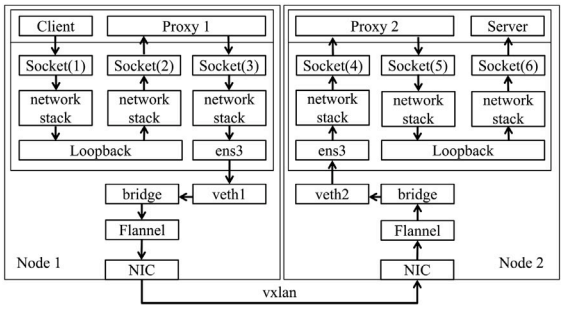
Processes	Value	Ratio			
		Without Istio		With Istio	
		Latency	Ratio	Latency	Ratio
Istio proxies	$T^3_{send}-T^2_{recv}$, $T^2_{send}-T^3_{recv}$, $T^5_{send}-T^4_{recv}$, $T^4_{send}-T^5_{recv}$	–	–	7.83 ms	25.83%
Kernel network stack	$T^2_{recv}-T^1_{send}$, $T^1_{recv}-T^2_{send}$, $T^6_{recv}-T^5_{send}$, $T^5_{recv}-T^6_{send}$	5.40 ms	32.82%	7.69 ms	25.40%
Applications	$T^1_{send}-T^1_{recv}$, $T^6_{send}-T^6_{recv}$	11.044 ms	67.18%	14.78 ms	48.77%
Total	–	16.44 ms	100%	30.31 ms ↑	100%



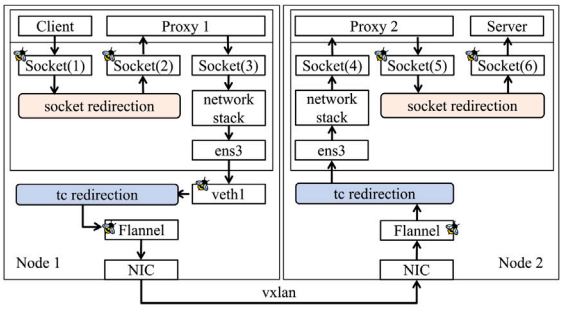
(a) Request for the service in the same node with the default implementation.



(b) Request for the service in the same node with our optimization.



(c) Request for the service in another node with the default implementation.



(d) Request for the service in another node with our optimization.

Fig. 4. Request packet transmission with and without our optimization.

25.85%, 25.4% and 48.77% of latency, respectively. The transmission delay consumed in the proxies of service meshes and applications is inherent and cannot be removed in a non-intrusive way due to the design of service meshes. Thus, in this study, we attempt to shorten the packet transmission path in kernel network stack to reduce the increased latency incurred by service meshes.

From the above observation, bypassing the network stack can reduce request latency by up to 25%. Istio proxies utilize *netfilter* and *iptables* in the kernel to define rules on packet forwarding, so that all the inbound and outbound traffic of a service instance have to pass through its proxy. *iptables* matches the forwarding rules of the packets sequentially. With the computational complexity of $O(n)$ for n rules, it may introduce additional delay due to more matching rules for packets. In this paper, we attempt to bypass some of the *iptables* to generate network shortcuts and reduce the request latency.

There already exist some studies on network bypassing, such as Data Plane Development Kit (DPDK) (Intel, 2014), NetMap (Rizzo, 2012) and AF_XDP (Kuhr and Carôt, 2020). However, the polling drivers of DPDK and Netmap increase CPU consumption, and DPDK and AF_XDP require dedicated drivers. eBPF offers an alternative for network stack bypassing in service meshes without any dedicated drivers in the event-triggered mode, featuring hot-pluggable, universal, and flexible capabilities. Cilium (2020) implements its own service mesh by deploying per-node eBPF proxies to accelerate underlying packet transmission. However, Cilium requires redeployment of the Kubernetes cluster and microservices. In addition, a failure on a per-node proxy will cause all services deployed on that node to fail. To reduce the failure impact of the per-node proxy and the cost of cluster redeployment, Xu Yizhou (2021) and Merbridge (2022) utilize eBPF to speed up the container network of Istio. However, they only focus the network optimization

within one node. When a large number of cross-node requests happen in large-scale microservice applications, their proposed optimization methods are unable to perform well.

After the optimization objectives and the problems of existing approaches, we attempt to optimize network performance in service meshes that satisfy the following requirements. (1) The optimization should be **non-intrusive** without any modifications to source codes of services and service meshes. (2) The optimization should be **general** by supporting service meshes with various Kubernetes CNI. (3) The optimization should be **efficient** in accelerating both the inter-pod and intra-pod service communication within service meshes.

4. Optimization

4.1. Overview

As analyzed in Section 3.3, the latency incurred in the kernel network stack is the primary factor in reducing request latency in service meshes. It is intuitive to employ AF_XDP (Kuhr and Carôt, 2020) and DPDK (Intel, 2014) to accelerate packet processing. However, when using DPDK and AF_XDP, developers must upgrade their underlying network drivers, such as the DPDK driver, which can be costly. Fortunately, eBPF allows us to make packets bypass the network stack without any intrusion in application codes. Specifically, eBPF hooks at the socket and tc (traffic control) layer are helpful for packets to bypass the network stack, making it an advantageous choice for reducing request latency by circumventing the network stack in the Linux kernel.

Fig. 4 illustrates the request packet transmission before and after deploying our optimization. Arrows indicate the transmission direction of requests. The white boxes are the processes that requests go through. The beige and blue boxes represent the optimization processes within and between pods, respectively. For intra-pod communication, requests can bypass the kernel network stack when transmitting between the proxy and its alongside service. For inter-pod communication, there are two cases. When the client and server are deployed in the same node, packets can bypass the underlying network stack and bridge. When the client and server are deployed in different nodes, packets can be forwarded early to the Flannel network interface and bypass the bridge. By shortening the packet transmission path, our optimization can reduce the request latency within service meshes.

4.2. Intra-pod optimization

Services in service meshes have to communicate with their proxies first when they connect to other services. As shown in Fig. 4, a service instance is always accompanied by a proxy in a pod. They communicate with each other through the Loopback Interface, which we call it as the intra-pod communication. In the intra-pod communication, we can accelerate the packet transmission by reducing the time for passing through the network stack and Loopback Interface.

4.2.1. sockops, sk_msg

In order to accelerate the intra-pod communication without any intrusion into the existing service meshes, we choose eBPF to implement it in the Linux kernel. eBPF has some hooks at kernel sockets and allows the attached user-defined programs to run when those hooks are triggered. Our work mainly focuses on two types of programs supported by eBPF, which are *sockops* and *sk_msg*. A *sockops* program can be triggered when a socket connection is established. Some of the sockets' parameters such as IP addresses and ports can be accessed as the connection information by the *sockops* program. The *sockops* programs can be regarded as a collector for socket connection information. An *sk_msg* program is attached to a map storing some specific sockets. When these sockets call *sendmsg* or *sendfile* system calls to send a file or a message, the attached *sk_msg* program will execute. *sk_msg* programs can parse the parameters of these system calls to obtain the connection information of the running socket. In addition, *sk_msg* programs support socket redirection between two sockets. For example, if a socket is redirected to another socket, its sending messages are directly written into the buffer of that socket rather than its own socket buffer, so messages can bypass the underlying kernel network stack, where the message would otherwise need to be transmitted between these two socket buffer.

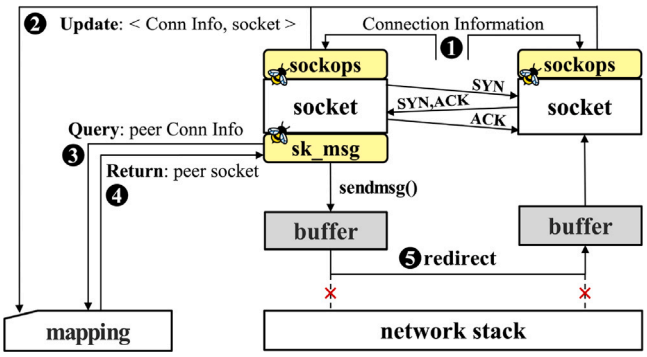


Fig. 5. The workflow of the intra-pod optimization.

4.2.2. Implementation

Fig. 5 and Algorithm 1 displays how the *sockops* and *sk_msg* programs work together to do socket redirection and accelerate intra-pod communication. A client is trying to send a request to a service. when the connection is established, the *sockops* program captures the established sockets as well as their connection information (source and destination IP addresses and ports). Then the sockops program constructs a key–value pair with the connection information as a key and the socket as a value, and updates it into a mapping table. After the connection is established, the established sockets send messages to each other and then trigger their *sk_msg* programs to run. The *sk_msg* programs take charge of bypassing the network stack in the kernel. Two sockets transmitting messages to each other are considered as a socket pair. For example, socket (1) and socket (2) in Fig. 4 constitute a socket pair, and socket (2) is the peer socket of socket (1) and vice versa.

The *sk_msg* program constructs the connection information of its peer socket and uses this information to query the mapping table, and then obtains the peer socket. Once the peer socket is found, the message to be sent is directly written into the buffer of the peer socket by the *sk_msg* program without passing through the underlying kernel network stack. As a result, messages are directly transmitted between socket pairs. In the meanwhile, headers of some network protocols, like TCP, will not be encapsulated into the transmitting messages and it reduces the amount of data to be transmitted, thereby reducing part of the transmission time.

Algorithm 1 Procedure of socket redirection

Require: socket s_i and its peer socket s_i', a transmitting message m_j between these two sockets, eBPF program *sockops* and *sk_msg*
  if s_i and s_i' in the same node **then**
    sockops monitors the establishing socket.
    sockops captures socket information of s_i and s_i'
    sk_msg finds s_i and s_i' as a socket pair
    sk_msg redirects message m_j between s_i and s_i'
  end if

It is important for the *sk_msg* program running in a socket to find its peer socket. To realize it, we first collect and observe the connection information of sockets in Istio as drawn in Fig. 4. For a common request, the client owns the *client_ip* as its pod IP. The server with the *server_ip* as the pod IP serves the service listening on (*service_ip: service_port*). The

Table 2
Connection information of sockets for a request in the Istio.

Socket	Source(ip:port)	Destination(ip:port)
(1)	client_ip:rdp1	service_ip:service_port
(2)	127.0.0.1:15001	client_ip:rdp1
(3)	client_ip:rdp2	server_ip:service_port
(4)	server_ip:15006	client_ip:rdp2
(5)	127.0.0.6:rdp3	server_ip:service_port
(6)	server_ip:service_port	127.0.0.6:rdp3

random port used in the established connection is denoted as *rdp*. The collected information is summarized in Table 2. The Loopback address (127.0.0.1 and 127.0.0.6) is used for the intra-pod communication in Istio. Istio typically utilizes port 15001 as the egress port and 15006 as the ingress port for a pod.

With the information from Table 2, for socket pair 1 ⟷ 2 and 5 ⟷ 6 in the intra-pod communication, the *sk_msg* can easily construct the peer socket of its own. Socket(1) and socket(2) can construct the connection information of its peer socket with the same key as "*client_ip:rdp1*". Socket(5) and socket(6) can easily find each other by exchanging their source and destination addresses and ports to construct the connection information of the peer. After finding the peer socket, the *sk_msg* program can correctly forward messages between socket pairs and bypass the network stack. It is worth noting that the *sockops* program is just used to update the running socket along with its connection information without the information construction of its peer socket. Since the *sockops* program is triggered by all the sockets in a node and it takes some time to construct the information of the peer socket, we have found that the information construction in the *sockops* program may introduce more time overhead than that in the *sk_msg* program.

4.3. Inter-pod optimization

The time spent on the inter-pod communication between two proxies also affects the request latency in service meshes. For example, messages have to pass through the network stack, the virtual network interface and the bridge in the kernel as shown in Fig. 4. We also try to bypass part of the kernel network stack to accelerate inter-pod packet transmission in service meshes.

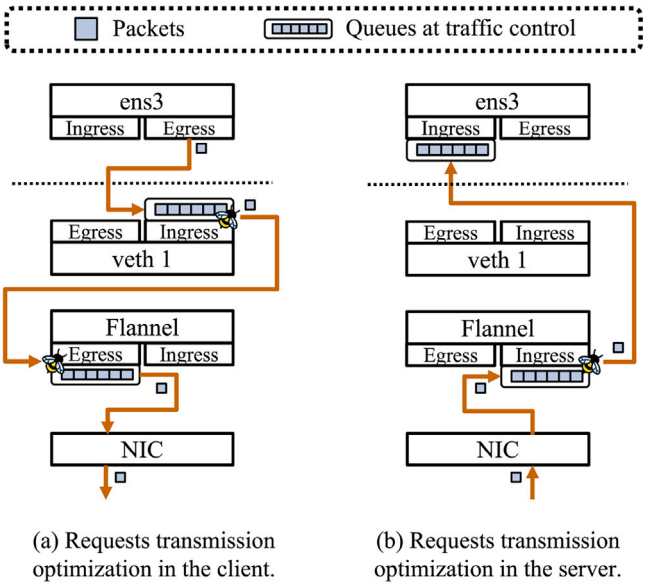
4.3.1. Cases within a node

We are first concerned about the cases where the client and the server are deployed in the same node in Fig. 4(b), where both socket (3) and (4) for the communication between proxies can be seen and captured in the kernel of the same node. Similar to the intra-pod optimization, the *sockops* program captures the established connection information. From Table 2, the *sk_msg* program can find out the peer socket with the same key as "*client_ip:rdp2*" for socket (3) and (4), then the *sk_msg* program can directly forward messages between socket (3) and (4) without passing through the underlying network stack and the bridge.

4.3.2. Cases across nodes

We then focus on the cases where the client and the server are deployed in different nodes in Fig. 4(d), as services are commonly deployed in different nodes in a distributed microservice system. To keep the service mesh non-intrusive, we utilize eBPF functionality on packet forwarding at the traffic control layer between the network interfaces.

The traffic control (tc) module implements policies for traffic control in Linux kernel, such as configuring different queuing rules for various queues (Vieira et al., 2020). Each NIC can build up a queue called *clsact* to process ingress and egress traffic. eBPF enables packets in the queue *clsact* of one NIC to be forwarded into that of another



(a) Requests transmission optimization in the client.

(b) Requests transmission optimization in the server.

Fig. 6. Inter-pod optimization across nodes.

NIC, so packets can bypass the transmission in some NICs or bridges. Both the inbound and outbound traffic of NIC can be processed in tc. As shown in Fig. 4(d) and detailed in Fig. 6, veth1, veth2, and ens3 are all virtual NICs. We first apply packet forwarding in advance from the ingress of veth1 to the egress of Flannel in node 1 with the eBPF helper function *bpf_redirect*, and from the ingress of Flannel to the ingress of ens3 (the peer of veth2) in node 2 with the helper function *bpf_redirect_peer*. As a result, in node 1, once the veth1 receives packets from its peer, ens3, it rapidly forwards them to the Flannel as outbound packets so that packets can bypass the network bridge. Similarly in node 2, once the Flannel receives packets which are sent to the server, it directly forwards them to the ens3, the peer of veth2, as inbound packets. The time for packets forwarding in bridges and veth2 then can be reduced.

To successfully execute the inter-pod optimization in service meshes, **the network symmetry and the changes in MAC (Media Access Control) addresses** during packet transmission have to be considered. On one hand, for the network symmetry, both the inbound and outbound traffic of a pod need to bypass the bridges, which means we need to deploy the tc programs to forward both the request and its response. On the other hand, since the queuing packets have been encapsulated in the ethernet header, their MAC addresses change when they are transmitted in the underlying network. In order to enable packets to be forwarded between two network interfaces, we employ a hash to record the MAC addresses for forwarding packets, and then we modify the MAC addresses of packets in advance.

5. Evaluation

To evaluate our optimization, we focus on the following three questions and answer them in our evaluation.

Q1: How effective is our optimization in improving performance in service meshes?

Q2: Compared to Cilium, a different implementation of the service mesh, does our optimization work better?

Q3: What is the overhead of our optimization on system resources?

5.1. Evaluation settings

We construct a Kubernetes platform with 5 virtual machines and run the service mesh Istio 1.14.3. Each virtual machine runs on a physical

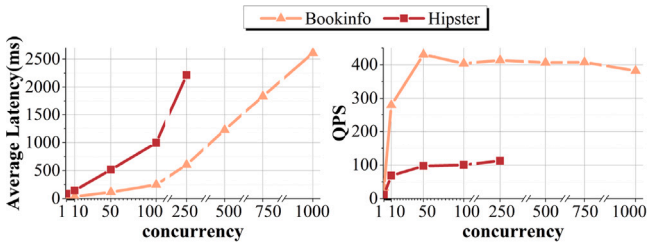


Fig. 7. The performance of Bookinfo and Hipster under different workloads.

machine separately, equipped with 16 GB RAM and a 32-core 2.80 GHz Intel Xeon Gold 6242 CPU. Each virtual machine runs on Ubuntu 18.04 OS and 5.10.85 Linux kernel. To run eBPF programs, *clang* and *LLVM* in version 10 are used to compile eBPF programs from source codes to bytecodes. *bpftool* with the same version of the Linux kernel is helpful to load the eBPF programs into the kernel and interact with eBPF maps. Fortio (Fortio, 2018) is used to generate user-defined query-per-second (QPS) loads to services and record their request latency.

5.1.1. Benchmarks

We deploy two open-source microservice benchmarks, Bookinfo (2022) and Google (2022) in our testbed. Bookinfo, provided by Istio, is a typical microservice benchmark in service meshes. Bookinfo serves as a single catalog entry of an online bookstore, which shows information about a book, including its description, details and comments. HipsterShop is a cloud-native application open-sourced by Google. It serves as a web-based e-commerce application and is composed of ten services written in different languages (e.g., Golang, Java, Python, C# and JavaScript). In our evaluation, Bookinfo is chosen as an example of HTTP applications, while HipsterShop is chosen as an example of gRPC applications. Each service owns 2–3 replicas.

5.1.2. Baselines

We use the following two state-of-the-art approaches as baselines.

- Cilium (2020) implements a service mesh with a per-node eBPF proxy to manage communication among services rather than a per-pod envoy proxy adopted in Istio. For our performance comparison, Cilium can be regarded as the fastest solution on network optimization in service meshes using eBPF.
- Merbridge (2022) is designed for Istio as a network acceleration solution. It also uses eBPF to bypass the kernel network stack and accelerate packet transmission.

5.1.3. Evaluation metrics

To evaluate the effectiveness of our optimization, we employ two widely-used metrics to represent service performance, which are the request latency and query-per-second (QPS). Request latency directly shows the real latency of user requests to benchmark applications under testing load. We mainly focus on the average as well as the 90th percentile (P90) of the request latency. QPS reflects the number of requests an application can handle in one second. Data on request latency and QPS is provided by the Fortio (Fortio, 2018). To evaluate the overhead of our optimization, CPU usage and memory usage are chosen as indicators of system resource usage.

5.2. Performance improvement

To demonstrate the effectiveness of our proposed optimization, Fortio is set to generate workloads for Bookinfo and HipsterShop under different concurrency. We first generate loads in different concurrency, then compare the results on average latency and QPS of Bookinfo and HipsterShop as shown in Fig. 7. As we can see, the maximum QPS served by the Bookinfo is around 400 and that of HipsterShop is around 100. Meanwhile, the curve of average latency of Bookinfo and HipsterShop becomes steeper when the concurrency reaches 100, A concurrency level of 100 serves as the threshold between low and high load. When the concurrency reaches 500, HipsterShop runs out of its capacity and can no longer serve properly. So the concurrency for benchmarks is set to be 1, 10, 100, 1000 for Bookinfo and 1, 10, 100 for HipsterShop. These experiments are executed in five cases, which are in the Kubernetes with Istio, Istio with Merbridge, Istio with only intra-pod optimization, Istio with only inter-pod optimization and Istio with our full optimization.[1] In our experiments, the target QPS is set to be unlimited and each experiment lasts for three minutes and is repeated five times.

Results of our experiments are drawn in Figs. 8 and 9. They show the average, P90 of the request latency and the QPS of Bookinfo and HipsterShop in benchmarks, where the error bars reflect the standard variance of the experiments repeated five times. In Figs. 8 and 9, the concurrency is denoted by c.

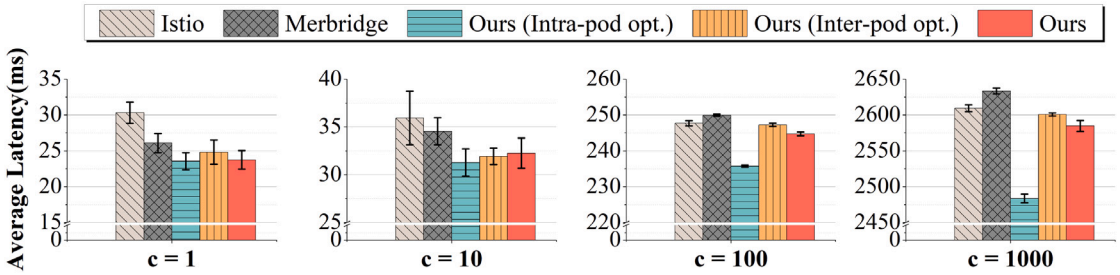
For Bookinfo, Merbridge accelerates packet transmission in Istio, reduces the average latency of Bookinfo by about 13%, 4% and improves QPS by 15% and 3.6% when the concurrency is low at 1 and 10, respectively. However, Merbridge does not perform well when concurrency reaches 100 and 1000. Under high concurrency, the average latency and QPS of Bookinfo when deploying Merbridge are similar to those with Istio, or even Merbridge introduces a higher latency to Bookinfo. In the meanwhile, our full optimization decreases the average latency of Bookinfo by 21%, 10%, 1% and 0.95%, improves QPS by about 27.8%, 11%, 1.2%, 0.95% at the concurrency of 1, 10, 100 and 1000, respectively, performing better than Merbridge.

For HipsterShop, our full optimization still outperforms Merbridge. Our full optimization improves by 9.7%, 4.5% and 1.4% on the average latency of HipsterShop and 10%, 4.4%, 1.4% on QPS when the concurrency is 1, 10 and 100, respectively, while Merbridge improves by about 6%, 4.8% and 0%, respectively. Since Merbridge changes the connection information of socket connection pairs, recording mapping from the original socket connection information to the new ones, Merbridge could not achieve better performance than ours.

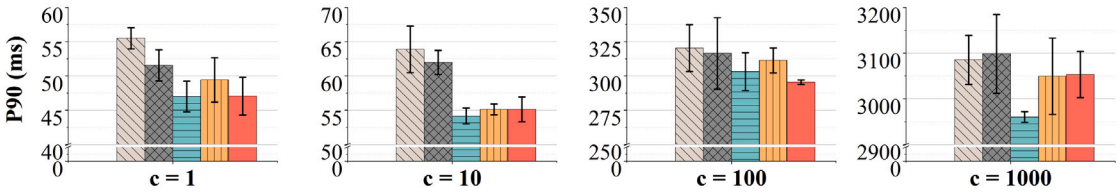
In order to better understand the effectiveness of our optimization, we conducted an ablation experiment and compared the performance between Istio, Istio with the intra-pod optimization, Istio with the inter-pod optimization and Istio with our full optimization. For Bookinfo, our full optimization improves its average request latency and QPS by 21%, 10%, 1% and 0.95% at the concurrency of 1, 10, 100 and 1000, while those with the inter-pod optimization are 18%, 11%, 1.8%, and 0.3%, and those with the intra-pod optimization are 22%, 13%, 4.8%, and 4.8%, respectively. For HipsterShop, our full optimization achieves improvement on latency and QPS of HipsterShop by about 9.7%, 4.5% and 1.4%, while those with the inter-pod optimization are 4.9%, 3.1%, 2.2%, and those with the intra-pod optimization are 8.6%, 5%, 1%. Istio with intra-pod optimization performs better than Istio with inter-pod optimization because the intra-pod optimization shortens the communication between a proxy and a service instance twice when the message is sent between two instances, while the inter-pod optimization only shortens once.

As we can see from Fig. 7, with the increase of the number of the concurrency, the queuing time of packets has a greater impact on the request latency, and the effect of our optimization and Merbridge becomes not obvious. Meanwhile, our optimization consumes more overhead associated with high concurrent accesses to the mappings used in the optimization between pods and within pods under high concurrency. Our optimization performs better on HTTP requests than we do on gRPC because of the optimization on the communication of

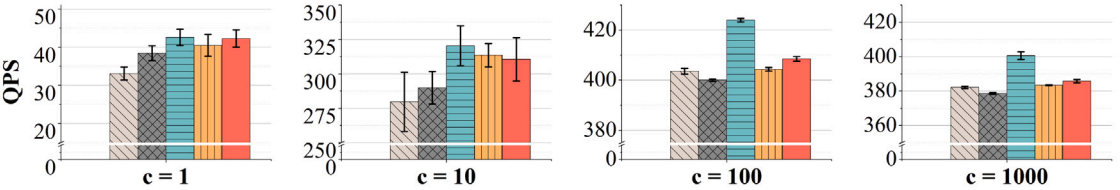
[1] Source codes are available at https://github.com/IntelligentDDS/ServiceMeshRedirect.



(a) Average response time of Bookinfo under different concurrency.

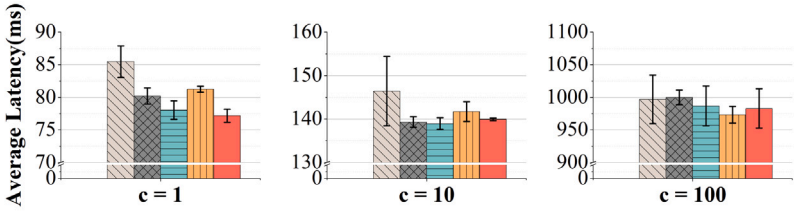


(b) P90 of Bookinfo under different concurrency.

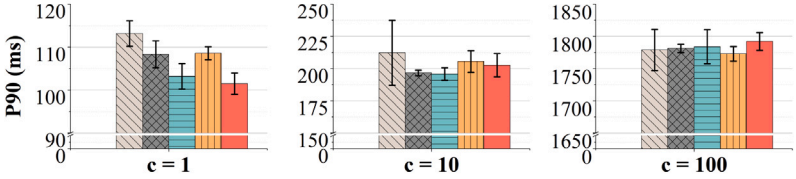


(c) QPS of Bookinfo under different concurrency.

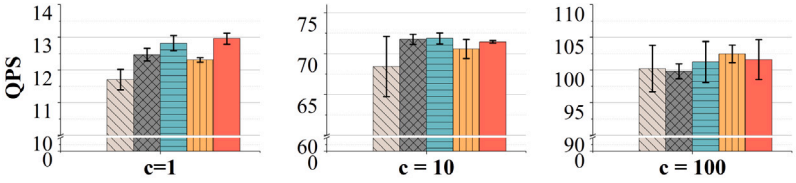
Fig. 8. Comparison results on service performance of Bookinfo when deploying Istio, Istio with Merbridge, Istio with only intra-pod optimization, Istio with only inter-pod optimization and Istio with our full optimization.



(a) Average response time of HipsterShop under different concurrency.



(b) P90 of HipsterShop under different concurrency.



(c) QPS of HipsterShop under different concurrency.

Fig. 9. Comparison results on service performance of HipsterShop with Istio, Istio with Merbridge, Istio with only intra-pod optimization, Istio with only inter-pod optimization and Istio with our full optimization.

Table 3
Comparison results on service performance of Bookinfo and HipsterShop when deploying Cilium, Istio and Istio with our full optimization.

Metrics	Settings	Bookinfo			HipsterShop		
		Istio	Istio(with opt.)	Cilium	Istio	Istio(with opt.)	Cilium
Average Latency	c = 10^0	30.3 ± 1.5 ms	23.7 ± 1.2 ms(1.22x)	18.9 ± 0.0 ms(**1.38x**)	85.5 ± 2.4 ms	77.2 ± 1 ms(1.09x)	73.1 ± 0.7 ms(**1.14x**)
	c = 10^1	35.9 ± 2.8 ms	32.3 ± 1.6 ms(**1.10x**)	33.8 ± 0.2 ms(1.06x)	146.4 ± 7.9 ms	140 ± 0.2 ms(**1.04x**)	161.2 ± 1.4 ms(0.9x)
	c = 10^2	247.7 ± 0.7 ms	244.7 ± 0.6 ms(**1.01x**)	327 ± 1.6 ms(0.68x)	997 ± 37.3 ms	983 ± 30.1 ms(**1.01x**)	1126 ± 13 ms(0.88x)
	c = 10^3	2.61 ± 0.005 s	2.58 ± 0.007 s(**1.01x**)	3.51 ± 0.012 s(0.66x)	–	–	–
QPS	c = 10^0	33.0 ± 1.7	42.2 ± 2.2(1.28x)	52.9 ± 0.3(**1.60x**)	11.7 ± 0.3	13.0 ± 0.2(1.107x)	13.7 ± 0.1(**1.111x**)
	c = 10^1	279.6 ± 21.5	310.6 ± 15.7(**1.11x**)	296.2 ± 1.9(1.06x)	68.4 ± 3.7	71.4 ± 0.2(**1.04x**)	61.9 ± 0.5(0.88x)
	c = 10^2	403.5 ± 1.1	408.5 ± 0.9(**1.01x**)	305.61 ± 1.5(0.76x)	100.2 ± 3.6	101.6 ± 3.0(**1.01x**)	88.6 ± 1.0(0.87x)
	c = 10^3	382.2 ± 0.6	385.8 ± 1.0(**1.01x**)	282.5 ± 1.0(0.74x)	–	–	–

the gRPC service itself. By comparing the P90 of requests in Figs. 8 and 9, our optimization and Merbridge both mainly improve the service performance of Bookinfo and HipsterShop by reducing their P90.

Answers for Q1: Our optimization performs better than Merbridge and improves the average request latency and QPS by up to 21% for Bookinfo and 10% for HipsterShop, mainly improving the latency of Bookinfo and HipsterShop for the first 90% of requests. Our optimization improves service performance under high concurrency less than that under low concurrency because of the increasing queuing time of packets and frequent concurrent access to the socket hash on our implementation.

5.3. Architecture comparison

Since Cilium implements the service mesh with eBPF in Linux kernel, we compare it with our optimization to better understand our performance. We run benchmarks with the same settings mentioned in Section 5.2 and present the results in Table 3.

From Table 3, we can observe that Cilium performs better than Istio and our full optimization at low concurrency. Cilium can achieve a 1.38x speedup for Bookinfo and a 1.14x speedup for HipsterShop, while our optimization only performs a 1.22x speedup for Bookinfo and a 1.09x speedup for HipsterShop. Cilium implements the service mesh with an in-kernel proxy per node, unlike Istio with per-pod proxies, so Cilium reduces the time overhead of going through the proxies repeatedly for packets. In addition, Cilium implements its underlying network of Kubernetes so that it can further reduce the overhead of the kernel network stack. These are the reasons why Cilium performs better than ours at low concurrency.

With the increase of concurrency, the enhancement of both our optimization and Cilium decreased. In the case of high concurrency, the per-node proxies have to process and redirect more requests, resulting in lower optimization performance than Istio. With the per-pod proxies, our optimization can achieve a better optimization effect than Cilium under higher concurrency.

Answer for Q2: Our optimization can achieve higher performance than Cilium under high concurrency, since per-node proxies used in Cilium must process all the requests in the local node.

5.4. Overhead

In order to evaluate the overhead introduced by our optimization, we keep running experiments on Bookinfo for three minutes with the concurrency as 100 and then we record the CPU and memory usage of each node once per second during the experiments. *sar* and *free* commands in Linux are used to capture the CPU and memory usage. The results are captured three times in two cases, when the nodes are running properly in Istio with and without our full optimization. The average results are shown in Fig. 10.

First, we focus on the consumed CPU resources. Our full optimization can reduce CPU consumption by 1.73% compared with the Istio, in total. Our full optimization reduces CPU usage in kernel by 0.5%, since packets bypass some of the kernel network stack. Moreover, our full optimization reduces 1.23% of the CPU consumption in the user space. This reduction comes from the fact that the application does not have to wait a long time for an upstream response to return because of the faster message transmission.

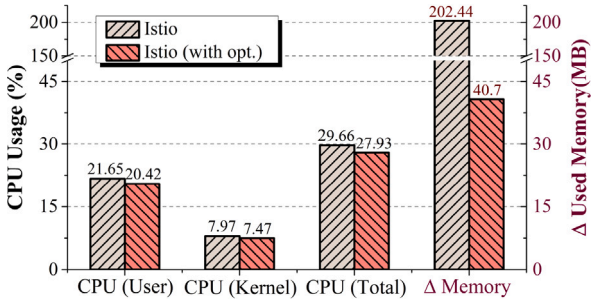


Fig. 10. Comparison results on CPU and memory usage of Istio with and without our full optimization in the same service deployment.

Then, we compare the memory used during the experiments. Since memory is likely to be used more and more while the node is working, we calculate the difference between the maximum and minimum of the used memory during the experiments as the memory usage. Observed from Fig. 10, about 161 MB (0.98%) more memory is used in Istio than that with our full optimization. Our optimization can reduce the memory used in packet copying between the network interface and the Linux kernel, thus reducing the memory replication.

Answer for Q3 : Our proposed optimization can reduce the overhead of CPU resources consumed by Istio since we can make packets bypass some of the kernel network stack. Meanwhile, we can reduce the used memory resources when copying packets between the network interface and Linux kernel.

5.5. Discussion and future work

The experimental results above show the effectiveness of our optimization. However, it can still be improved and extended in future work. Inspired by XRP (Zhong et al., 2022), each socket can be stored in a separate space to reduce the time overhead of concurrent access to a hash, thus improving the service performance at high concurrency. In addition, we plan to further optimize the packet transmission path by sharing established socket connections between pods of the same service, thereby reducing the time overhead of passing through the network stack during socket establishing.

Table 4 shows the comparison among Cilium, Merbridge and our optimization in detail. All the optimization based on eBPF requires the root privilege. Cilium performs well on network optimization of service meshes at low concurrency. However, it needs to change the architecture from the original Kubernetes underlying network to Cilium, which limits the generality of Cilium. In addition, a proxy is deployed on each node in Cilium, thus the failure on a proxy has a greater impact on the overall communication in a node than Merbridge and our optimization. Merbridge performs worse than ours since it does not accelerate packet

transmission across nodes and spends more time on the maintenance of the mapping between its established sockets. Our optimization can accelerate packet transmission for developer-specified services without interfering with other communication between multiple containers in a pod. Furthermore, if a fault occurs when the *sk_msg* program queries for the peer socket, the packet transmission is reverted to the original network path, which guarantees the normal communication among services to a certain degree. Our optimization is also pluggable and easy for software developers and operators to install and uninstall.

6. Conclusion

In this paper, we study the packet transmission path with and without service mesh deployed. To avoid making any code intrusions to applications and proxies of the service mesh, we choose to shorten the packet transmission path to improve the application performance. We use eBPF to optimize request latency in a service mesh. Packets are forwarded directly between sockets so they do not need to pass through the kernel network stack. For cross-node communication, we shorten the request latency by directly forwarding packets from one network interface to another at the traffic control layer to bypass the bridge. Practical evaluations demonstrate that our proposed method can significantly optimize latency in Istio in a non-intrusive way, and reduce 1.73% of the CPU consumption and 0.98% of the memory.

In our future work, we plan to improve socket query performance by storing each socket in a separate space to improve our optimization performance on concurrent requests. Moreover, by sharing established socket connections between pods of the same service, we expect to further optimize the packet transmission path to reduce the time overhead when packets pass through the network stack to establish a new socket connection.

CRediT authorship contribution statement

Wanqi Yang: Conceptualization, Methodology, Software, Verification, Writing – original draft, Writing – review & editing. **Pengfei Chen:** Conceptualization, Writing – original draft, Writing – review & editing. **Guangba Yu:** Writing – original draft, Writing – review & editing. **Haibin Zhang:** Resources, Investigation. **Huxing Zhang:** Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

I have shared the link to my code in the manuscript.

Acknowledgments

We greatly appreciate the insightful feedback from the anonymous reviewers. The research is supported by the National Key Research and Development Program of China (2019YFB1804002), the National Natural Science Foundation of China (No.62272495), the Guangdong Basic and Applied Basic Research Foundation (No.2023B1515020054). This research is also sponsored by Alibaba project (No.20925056).

Table 4
Comparison among Cilium, Merbridge and our optimization.

	Cilium	Merbridge	Ours
Proxy Deployment	per-node	per-pod	per-pod
Impact of proxy breakdown	node	pod	pod
Architecture Supporting	–	Istio	Istio
Underlying Network	Cilium	iptables	iptables
Accelerate within a node	Yes	Yes	Yes
Accelerate across nodes	Yes	No	Yes
Effectiveness	good	good	better

References

Abranches, M., Michel, O., Keller, E., Schmid, S., 2021. Efficient network monitoring applications in the kernel with eBPF and XDP. In: 2021 IEEE conference on network function virtualization and software defined networks. NFV-SDN, pp. 28–34.

AG, S., 2022. Annual APIs and Integration Report - 2022, https://www.softwareag.com/en_corporate/resources/asset/ar/integration-api/apis-integration-microservices-report.html.

Ahmed, Z., Alizai, M.H., Syed, A.A., 2018. InKeV: In-kernel distributed network virtualization for DCN. SIGCOMM Comput. Commun. Rev. 46 (3), 1–6.

Baidya, S., Chen, Y., Levorato, M., 2018. eBPF-based content and computation-aware communication for real-time edge computing. In: IEEE INFOCOM 2018-IEEE conference on computer communications workshops. INFOCOM WKSHPS, IEEE, pp. 865–870.

BCC, 2022. BCC: Tools for BPF-based linux IO analysis, networking, monitoring, and more. https://github.com/iovisor/bcc.

Bernstein, D., 2014. Containers and cloud: From lxc to docker to kubernetes. IEEE Cloud Comput. 1 (3), 81–84.

Bookinfo, 2022. Bookinfo. https://istio.io/latest/docs/examples/bookinfo/.

bpftrace, 2018. bpftrace: High-level tracing language for Linux eBPF. https://github.com/iovisor/bpftrace.

Brondolin, R., Santambrogio, M.D., 2020. A black-box monitoring approach to measure microservices runtime performance. ACM Trans. Archit. Code Optim. 17 (4), 1–26.

Calcote, L., Butcher, Z., 2019. Istio: Up and Running: Using a Service Mesh to Connect, Secure, Control, and Observe. O'Reilly Media.

Calico, 2023. Calico : Cloud native networking and network security. https://github.com/projectcalico/calico.

Choe, Y., Shin, J.-S., Lee, S., Kim, J., 2020. eBPF/XDP based network traffic visualization and dos mitigation for intelligent service protection. In: International Conference on Emerging Internetworking, Data & Web Technologies. Springer, pp. 458–468.

Cilium, 2020. Cilium: eBPF-based networking, observability, and security. https://cilium.io/.

Cinque, M., Della Corte, R., Pecchia, A., 2022. Micro2vec: Anomaly detection in microservices systems by mining numeric representations of computer logs. J. Netw. Comput. Appl. 208, 103515.

Corbet, J., 2014. Extending extended BPF. Linux Weekly News.

eBPF, 2022. eBPF. https://ebpf.io.

Envoy, 2019. Envoy: an open source edge and service proxy, designed for cloud-native applications. https://www.envoyproxy.io/.

Facebook, 2018. Katran. https://github.com/facebookincubator/katran.

Flannel, 2014. Flannel: a network fabric for containers, designed for kubernetes. https://github.com/flannel-io/flannel.

Fortio, 2018. Fortio load testing library, command line tool, advanced echo server and web UI in go (golang). https://github.com/fortio/fortio.

Google, 2022. HipsterShop: Sample cloud-native application with 10 microservices showcasing Kubernetes, Istio, gRPC and OpenCensus. https://github.com/GoogleCloudPlatform/microservices-demo.

Hauser, F., Häberle, M., Merling, D., Lindner, S., Gurevich, V., Zeiger, F., Frank, R., Menth, M., 2023. A survey on data plane programming with P4: Fundamentals, advances, and applied research. J. Netw. Comput. Appl. 212, 103561.

Intel, D., 2014. Data plane development kit.

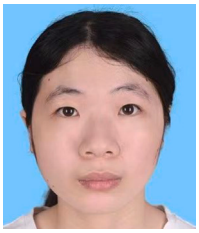
Istio, 2022. Istio: Simplify observability, traffic management, security, and policy with the leading service mesh. https://istio.io.

Jadin, M., De Coninck, Q., Navarre, L., Schapira, M., Bonaventure, O., 2022. Leveraging eBPF to make TCP path-aware. IEEE Trans. Netw. Serv. Manag. 19 (3), 2827–2838.

Jouet, S., Pezaros, D.P., 2017. BPFabric: Data plane programmability for software defined networks. In: 2017 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS). IEEE, pp. 38–48.

Kim, T., Ng, D.M., Gong, J., Kwon, Y., Yu, M., Park, K., 2023. Rearchitecting the TCP stack for I/O-Offloaded content delivery. In: 20th USENIX Symposium on Networked Systems Design and Implementation. NSDI 23, USENIX Association, Boston, MA, pp. 275–292.

Kubernetes, 2022. Kubernetes: Production-grade container orchestration. https://kubernetes.io/.
Kuhr, C., Carôt, A., 2020. eXpress data path kernel objects for real-time audio streaming optimization. https://lac2020.sciencesconf.org/307835/document.
Lee, C., Yoshitani, R., Hirotsu, T., 2022. Enhancing packet tracing of microservices in container overlay networks using EBPF. In: Proceedings of the 17th Asian Internet Engineering Conference. AINTEC '22, Association for Computing Machinery, New York, NY, USA, pp. 53–61.
Li, W., Lemieux, Y., Gao, J., Zhao, Z., Han, Y., 2019. Service mesh: Challenges, state of the art, and future research opportunities. In: 2019 IEEE International Conference on Service-Oriented System Engineering. SOSE, IEEE, pp. 122–1225.
libbpf, 2020. libbpf: a userspace library for loading and interacting with bpf programs. https://github.com/libbpf/libbpf.
Mahalingam, M., Dutt, D., Duda, K., Agarwal, P., Kreeger, L., Sridhar, T., Bursell, M., Wright, C., 2014. Virtual extensible local area network (VXLAN): A framework for overlaying virtualized layer 2 networks over layer 3 networks. Technical Report.
McCanne, S., Jacobson, V., 1993. The BSD packet filter: A new architecture for user-level packet capture. In: Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings. USENIX '93, USENIX Association, p. 2.
Merbridge, 2022. Merbridge. https://github.com/merbridge/merbridge. (Accessed 6 February 2023).
Miano, S., Bertrone, M., Risso, F., Tumolo, M., Bernal, M.V., 2018. Creating complex network services with ebpf: Experience and lessons learned. In: 2018 IEEE 19th International Conference on High Performance Switching and Routing. HPSR, IEEE, pp. 1–8.
Miano, S., Chen, X., Basat, R.B., Antichi, G., 2023. Fast in-kernel traffic sketching in EBPF. SIGCOMM Comput. Commun. Rev. 53 (1), 3–13.
Nam, T., Kim, J., 2017. Open-source IO visor eBPF-based packet tracing on multiple network interfaces of Linux boxes. In: 2017 International Conference on Information and Communication Technology Convergence. ICTC, IEEE, pp. 324–326.
Purdy, G.N., 2004. Linux Iptables Pocket Reference: Firewalls, NAT & Accounting. O'Reilly Media, Inc..
Qi, S., Monis, L., Zeng, Z., Wang, I.c., Ramakrishnan, K.K., 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. Association for Computing Machinery, pp. 780–794.
Qi, S., Zeng, Z., Monis, L., Ramakrishnan, K.K., 2023. MiddleNet: A unified, high-performance NFV and middlebox framework with eBPF and DPDK. IEEE Trans. Netw. Serv. Manag. 1.
Rizzo, L., 2012. netmap: A novel framework for fast packet I/O. In: 2012 USENIX Annual Technical Conference. USENIX ATC 12, USENIX Association, pp. 101–112.
Scholz, D., Raumer, D., Emmerich, P., Kurtz, A., Lesiak, K., Carle, G., 2018. Performance implications of packet filtering with linux eBPF. In: 2018 30th International Teletraffic Congress. ITC 30, Vol. 1, IEEE, pp. 209–217.
Shen, J., Zhang, H., Xiang, Y., Shi, X., Li, X., Shen, Y., Zhang, Z., Wu, Y., Yin, X., Wang, J., Xu, M., Li, Y., Yin, J., Song, J., Li, Z., Nie, R., 2023. Network-centric distributed tracing with DeepFlow: Troubleshooting your microservices in zero code. In: Proceedings of the ACM SIGCOMM 2023 Conference. Association for Computing Machinery, pp. 420–437.
Srirama, S.N., Adhikari, M., Paul, S., 2020. Application deployment using containers with auto-scaling for microservices in cloud environment. J. Netw. Comput. Appl. 160, 102629.
tcpdump, 2022. tcpdump: a powerful command-line packet analyzer. https://www.tcpdump.org/.
Vieira, M.A.M., Castanho, M.S., Pacífico, R.D.G., Santos, E.R.S., Júnior, E.P.M.C., Vieira, L.F.M., 2020. Fast packet processing with EBPF and XDP: Concepts, code, challenges, and applications. ACM Comput. Surv. 53 (1), 1–36.
Wan, X., Guan, X., Wang, T., Bai, G., Choi, B.-Y., 2018. Application deployment using microservice and docker containers: Framework and optimization. J. Netw. Comput. Appl. 119, 97–109.
Xhonneux, M., Duchene, F., Bonaventure, O., 2018. Leveraging ebpf for programmable network functions with ipv6 segment routing. In: Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies. Association for Computing Machinery, pp. 67–72.
Xu Yizhou, G.R., 2021. Accelerate Istio-CNI with ebpf. Technical Report, IstioCon 2021.
Yu, G., Chen, P., Zheng, Z., 2019. Microscaler: Automatic scaling for microservices with an online learning approach. In: ICWS 2019. IEEE, pp. 68–75.
Zhong, Y., Li, H., Wu, Y.J., Zarkadas, I., Tao, J., Mesterhazy, E., Makris, M., Yang, J., Tai, A., Stutsman, R., Cidon, A., 2022. XRP: In-kernel storage functions with eBPF. In: 16th USENIX Symposium on Operating Systems Design and Implementation. OSDI 22, USENIX Association, pp. 375–393.
Zhou, Y., Wang, Z., Dharanipragada, S., Yu, M., 2023. Electrode: Accelerating distributed protocols with eBPF. In: 20th USENIX Symposium on Networked Systems Design and Implementation. NSDI 23, USENIX Association, Boston, MA, pp. 1391–1407.



Wanqi Yang received the B.S. degree in Computer Science and Technology from Sun Yat-Sen University, Guangzhou, China. She is currently working toward the M.S. degree in Computer Science and Technology with School of Computer Science and Engineering, Sun Yat-Sen University. Her research interests include eBPF, cloud computing and computer network.

Pengfei Chen is currently an associated professor in the School of Computer Science and Engineering of Sun Yat-sen University. Meanwhile, he is a Ph.D. advisor. Dr. Chen graduated from the department of computer science of Xi'an Jiaotong University with a Ph.D. degree in 2016. He is interested in distributed systems, AIOps, cloud computing, Microservice and BlockChain. Especially, he has strong skills in cloud computing. So far, Dr. Chen has published more than 60 papers in some international conferences including ACM/IEEE ASE, ACM/IEEE ICSE, IEEE INFOCOM, WWW and journals including IEEE TDSC, IEEE TNNLS, IEEE TR, IEEE TSC. He serves as of program committee member of multiple conferences and reviewers of some internal journals such as IEEE TPDS, ACM TOSE, IEEE TDSC, IEEE TC.

Guangba Yu received his master degree from Sun Yat-Sen University, China, in 2020. He is now a phd student at School of Computer Science and Engineering with Sun Yat-Sen University, China. His current research areas include distributed system, cloud computing, and AI driven operations. So far, he has published more than 20 papers in some international conferences including ACM/IEEE ASE, ACM/IEEE ICSE, WWW and journals including IEEE TDSC, IEEE TNNLS, IEEE TSC.

Haibin Zhang is currently a technical expert in the Observable Team of Cloud Native Application Platform of Alibaba Cloud Intelligence Group, focusing on observable and service governance technologies based on eBPF. He received his master's degree from Hefei University of Technology in 2015.

Huxing Zhang is currently a Staff Engineer at Alibaba Cloud, he received his Master's degree from Shanghai Jiaotong University. His research interests include cloud native, microservices, and observability.