arXiv:2305.05455v3 [cs.NI] 4 Jun 2024

ONCache：一种基于缓存的低开销容器覆盖网络

Shengkai Lin[1], Shizhen Zhao[1],Peirui Cao[1],Xinchi Han[1],Quan Tian[2], Wenfeng Liu[2],Qi Wu[2],Donghai Han[2],和 Xinbing Wang[ll]上海交通大学， [2]*VMware*

摘要

近年来，容器得到了广泛应用。虽然容器简化并加速了应用程序开发，但现有的容器网络技术要么开销显著，影响分布式应用程序的性能；要么缺乏灵活性或兼容性，阻碍了其在生产环境中的广泛部署。

我们仔细分析了覆盖网络的内核数据路径，量化数据路径各个部分所消耗的时间，并识别出覆盖网络相较于裸机的额外开销 。我们观察到，这种额外开销会在数据包之间产生重复结果，这启发我们在覆盖网络中引入缓存。

我们设计并实现了 ONCache（**覆盖网络缓存**），这是一种基于缓存的容器覆盖网络，旨在消除额外开销，同时保持灵活性和兼容性。我们使用扩展伯克利数据包过滤器（扩展伯克利包过滤器）实现了 ONCache，仅需 524 行代码，并将其集成为 Antrea 的插件。借助 ONCache，容器能够实现媲美裸机的网络性能。与标准覆盖网络相比，ONCache 将 TCP 的吞吐量和请求-响应事务速率分别提升了 12% 和 36%（UDP 分别提升了 20% 和 34%），同时显著降低了每数据包 CPU 开销。热门分布式应用也能从 ONCache 中受益。

1 引言

容器 [9, 21] 因其灵活性和轻量特性，在分布式应用程序部署中日益普及。通过将应用程序及其所有依赖项和配置文件打包到一个容器镜像中，容器实现了一次构建，随处运行。此外，容器利用主机操作系统内核，而不是模拟整个操作系统，因此比虚拟机更轻量。Kubernetes 等容器编排工具 [35],Docker Swarm [22], 等则通过提供

技术	性能	灵活性	兼容性
Host	✓	✗	✓
网桥	✓	✗	✓
Macvlan	✓	✗	✓
IPvlan	✓	✗	✓
SR-IOV [64]	✓	✗	✓
覆盖网络	✗	✓	✓
Falcon [37]	✗	✓	✓
Slim [74]	✓	✓	✗
ONCache	✓	✓	✓

表 1：比较容器网络技术。

自动部署、自动扩缩容和自动高可用等功能 [8]。

容器网络在实现容器间通信和支持分布式应用方面发挥着关键作用。理想情况下，容器网络应在保持灵活性和兼容性的同时，提供低开销、高性能的网络服务。然而，正如表 1. 所示，现有的容器网络都无法完全满足这些期望。

损害灵活性。 主机网络中的容器共享主机 IP 地址，并且必须协调主机上的端口，从而限制了部署灵活性。网桥网络或设备虚拟化技术（例如 Macvlan/IPvlan/SR- IOV [64]）允许容器拥有自己的 IP 地址，并将容器数据包直接转发到底层网络。然而，这些网络要求所有主机位于同一二层网络中，或者底层网络能够路由容器数据包，从而对容器放置、迁移或网络配置施加限制。

性能不佳且开销高。 采用隧道技术（例如 VXLAN [41]）的容器覆盖网络将容器与底层网络完全解耦，使容器 IP 地址的配置不受底层网络的限制。尽管如此，这种方式仍会产生显著开销 [37, 38, 65, 74]。我们的实验表明，与裸机相比，基于隧道的覆盖网络中单个 TCP 流的吞吐量约低 11%，而

TCP 请求-响应事务速率约低 29%。此外，在两项实验中，CPU 利用率都高得多。尽管有些技术 [36,37,59] 试图通过将入站数据包处理分散到多个 CPU 核心来提升覆盖网络的性能，但覆盖网络的处理开销仍然很大。

损害兼容性。 Slim [74] 及其后续研究 [12,39,40] 通过套接字替换实现覆盖网络，使容器化应用程序能够使用主机的套接字。虽然它们在覆盖网络中保持了较低的开销和一定程度的灵活性，但它们本身并不支持非连接型协议（例如 UDP、ICMP）或容器实时迁移。此外，它们可能与底层网络中基于隧道标头的策略发生冲突。

为了同时实现高性能、灵活性和兼容性，我们设计并实现了 ONCache（**O**verlay**N**etwork **Cache**），这是一种采用基于缓存、低开销快速路径的容器覆盖网络。ONCache 可以无缝集成到现有的标准覆盖网络中，并已在 Antrea [1]和 Flannel [28]上进行了测试。

我们的主要贡献总结如下：

1. 我们仔细分析了常见容器覆盖网络（Antrea 和 Cilium）以及裸机的内核数据路径，并量化了各部分开销所消耗的时间。分析揭示了容器覆盖网络产生的额外开销。然而，这些开销分散在内核数据路径中，因此在保留必要功能的同时降低开销面临重大挑战。（§2.2）2. 我们发现了额外开销中的不变性，这表明额外开销的处理结果在不同数据包之间具有重复性。这一特性启发我们在覆盖网络中使用缓存。（§2.4）3. 我们为容器覆盖网络提出了一种基于缓存的快速路径。该快速路径在不损失灵活性和兼容性的前提下绕过额外开销。（§3）4. 我们使用微基准测试和常见分布式应用对 ONCache 进行了评估。结果表明，与标准覆盖网络相比，ONCache 显著提升了吞吐量（使用 iperf3 时提高 12%）、延迟（事务速率提高 36%），并降低了中央处理器利用率（每笔事务降低 26%），其性能接近裸机。Memcached、PostgreSQL 和 Nginx 等应用均受益于 ONCache。（§4）

ONCache 的源代码已公开，可在以下地址获取：https://github.com/nothepeople/ONCache。本研究不涉及任何伦理问题。

2 背景与动机

我们分析主机间的容器网络，其中覆盖网络以出色的灵活性和兼容性脱颖而出。在量化覆盖网络的详细开销并探讨近期旨在降低这些开销的工作后，我们阐述了 ONCache 的设计动机。

2.1 跨主机容器网络

容器网络在促进容器之间以及容器与外部世界之间的通信方面发挥着关键作用。下文重点讨论跨主机容器网络，这类网络根据应用程序 和物理网络所使用的 IP 地址进行分类。

两者都使用主机 IP 地址。 主机网络属于此类。主机网络中的容器共享主机的网络命名空间，并使用主机的 IP 地址。主机网络的性能可与裸机相媲美（即应用程序直接在主机上运行）[65]。然而，主机网络要求协调同一主机上各容器使用的端口，从而大幅限制了应用程序的灵活性。因此，主机网络在生产环境中很少使用 [66]。

两者都使用容器 IP 地址。 桥接网络和设备虚拟化技术（如 Macvlan、IPvlan、SR-IOV）属于此类。在这类网络中，每个容器都会分配到唯一的 IP 地址，并与主机命名空间隔离。在桥接网络中，所有容器都连接到 Open vSwitch（OVS） [1],网桥 [28],或类似实体，并依赖它们将数据包转发至物理网络。采用设备虚拟化技术的网络会在每个物理接口上模拟出多个具有不同 IP 地址的虚拟接口，每个虚拟接口均连接到一个容器。这些网络的数据路径简单，因此性能与裸机相近。

然而，由于这些方法在物理网络中使用容器 IP 地址，因此灵活性较差。要路由容器 IP 地址，必须 (1) 将所有主机置于同一个二层网络中，或 (2) 在物理网络中安装并管理针对容器 IP 地址的路由规则。这些要求限制了容器放置、迁移或网络配置。

应用使用容器 IP 地址，而物理网络使用主机 IP 地址。覆盖网络属于这一类。在覆盖网络中，每个容器都有自己的 IP 地址，容器数据包在传输到物理网络之前会通过隧道协议（例如 VXLAN [41] 或 GENEVE [29]）进行封装。覆盖网络将容器与物理网络完全解耦，从而确保两者的灵活性。然而，隧道会带来额外开销并损害性能（见 §2.2）。

要点 #1 覆盖网络将应用与物理网络解耦，使应用部署无需考虑物理网络配置，反之亦然。鉴于覆盖网络具有卓越的灵活性，我们专注于分析和改进覆盖网络。

	出站					入站				
数据路径	开销类型	Antrea	Cilium	BM	**Ours**	开销类型	Antrea	Cilium	BM	**Ours**
应用程序网络协议栈	skb 分配	1505	1566	1461	**1509**	skb 释放	715	818	780	**714**
	连接跟踪	778	0	788	**763**	连接跟踪	616	0	600	**592**
	netfilter	0	0	305	**0**	netfilter	0	0	173	**0**
	其他	423	560	547	**519**	其他	838	1016	979	**982**
Veth 对*	网络命名空间遍历	562	594		**489**	网络命名空间遍历	400			
扩展伯克利包过滤器*	eBPF		1513		**511**	eBPF		1429		**289**
Open vSwitch*	连接跟踪	872				连接跟踪	758			
	流匹配	354				流匹配	308			
	动作执行	92				动作执行	66			
VXLAN 网络协议栈*	连接跟踪	0	471			连接跟踪	0	271		
	netfilter	667	421			netfilter	466	303		
	路由	50	468			路由	294	554		
	其他	319	127			其他	619	444		
链路层	链路层	1858	1763	1799	**1700**	链路层	2790	2848	2800	**2737**
Sum		7479	7483	4900	**5491**		7869	7683	5332	**5315**
延迟 （μ秒）		22.97	23.15	16.57	**17.49**		22.97	23.15	16.57	**17.49**

表2： 不同网络的开销分解。除最后一行的端到端延迟外，所有值的单位均为纳秒。*BM* 表示裸机。我们的方法（即 *ONCache*）在 *§3* 中提出，并在 *§4*中评估。“*”表示相较于裸机的额外开销。由于测量工具的限制，误差约为 *200* 纳秒。

2.2 覆盖网络开销分析

先前的研究 [37,38,65,74] 已表明覆盖网络的性能较差。我们进一步量化了两种常见覆盖网络的开销：Antrea [1] (v1.9.0) 和 Cilium [13] (v1.12.4)，并与裸机进行比较。

我们按照以下步骤分析网络。首先，借助火焰图对出站和入站数据路径进行定性分解 [27],，火焰图提供了数据路径上完整的函数调用栈。接着，我们从火焰图中找出代表开销各个部分的确切函数（及其调用栈）。最后，我们使用扩展伯克利包过滤器（eBPF）测量每个函数的执行时间，进而计算开销各部分的执行时间。详细的分析方法见附录A。

表 2 展示了分解后的开销以及开销各部分的执行时间。每项所列的执行时间均为 1 秒 1 字节 TCP 请求-响应（RR）测试中所有计时样本的平均值。我们主要分析裸机和 Antrea 中的开销（“*”表示相较于裸机增加的额外开销），并讨论 Cilium 中不同的开销。我们的方法 ONCache 将在 §4中分析。

应用程序网络协议栈。 应用程序网络协议栈直接与应用程序交互。在出口路径上，应用程序将数据写入套接字。随后，应用程序网络协议栈为数据分配套接字缓冲区（即 skb），并逐层封装数据包。相反，在入口路径上，网络协议栈会解封装标头并释放套接字缓冲区。此外，netfilter [47] 会在入口和出口路径上过滤数据包。连接跟踪 [18],作为 netfilter 的一个组件，会跟踪连接状态（并非 TCP 连接，而是由例如五元组定义的连接：源 IP 地址、源端口、目的 IP 地址、目的端口和传输协议），并支持有状态过滤器应对探测和拒绝服务攻击。netfilter 和连接跟踪的执行取决于系统配置。

Veth 对*。容器在各自的网络命名空间中运行应用程序，并使用虚拟以太网设备对 [69] 与主机命名空间连接。数据包经过虚拟以太网设备对时会产生开销，其中包括发送端虚拟以太网设备上的发送队列开销，以及接收端虚拟以太网设备上的软件中断调度开销。在 Cilium 中，虽然可以通过 bpf（Berkeley Packet Filter）重定向 [71],来减轻数据包在入口路径上经过虚拟以太网设备对时产生的开销，但在出口路径上 [17]仍然存在这种开销。

OVS*。OVS 会过滤数据包，并在容器与主机接口之间进行主机内路由。这些过滤和路由操作由流描述。OVS 的开销可分为连接跟踪、流匹配和动作执行。尽管 OVS 使用缓存来加快流匹配速度， [53],连接跟踪仍会消耗大量中央处理器时间。Cilium 使用扩展伯克利包过滤器实现了与 OVS 类似的功能 [16]。然而，扩展伯克利包过滤器的总体执行时间与 OVS 相近。

VXLAN[1] 网络协议栈*。VXLAN 网络协议栈负责对 VXLAN 数据包执行出站和入站路由，并对数据包进行外层首部的封装或解封装（包括 VXLAN报头、UDP报头、IP 首部和 MAC 首部）。在 Antrea 中，VXLAN 路由由 OVS 加速，因此路由开销显著降低。此外，在此阶段，连接跟踪会跟踪连接，而 netfilter 会过滤数据包。

链路层。 在出站路径上，链路层将数据包排队并发送到网络；在入站路径上，它分配套接字缓冲区，并接收来自网络的数据包。

[1]在以下讨论中，我们以 VXLAN 为例。对其他隧道协议的分析也类似。

要点 **#2** 覆盖网络中的额外开销 包括虚拟以太网设备对、OVS（或类似实体）以及 VXLAN 网络协议栈，会产生显著开销并降低网络性能。由于这些开销分散在数据路径的各个部分，在保留必要功能的同时降低开销是一项重大挑战。

2.3 覆盖网络相关研究

近期有一些工作致力于提升覆盖网络的性能。

套接字替换。 Slim [74] 提出了一种套接字替换机制，可提高容器覆盖网络的性能。该机制在主机命名空间中建立 TCP 连接，并使用主机命名空间中的 TCP 套接字替换容器中的 TCP 套接字。尽管提高了 TCP 性能，Slim 仍存在明显的局限性。首先，它不支持 UDP 和 ICMP 等非连接型协议，也不支持容器实时迁移，因此限制了其兼容性。其次，Slim 的数据包并非隧道数据包，这会影响基于隧道头部进行匹配的底层网络策略。此外，在连接建立期间，Slim 必须为服务发现建立覆盖连接，导致连接建立时间延长。一些后续研究 [12, 39, 40]沿用了 Slim 的思路，并改进了连接建立时间。然而，它们仍然存在前述的两个缺陷。

CPU 负载均衡。Linux 扩展技术，例如接收数据包引导（RPS）和接收流引导（RFS）（以及对应的硬件技术接收端扩展（RSS）和加速型 RFS（aRFS））[59],通过将入口数据包处理分散到多个 CPU 核心上，并尝试降低缓存未命中率，来提升性能。Falcon [37] 和 mFlow [36] 探索了将数据包处理分散到多个核心上的不同策略。然而，它们以更高的 CPU 利用率为代价提升了网络性能。此外，它们仅对入口路径起作用，如果瓶颈位于出口路径，则无法进一步提升性能。

要点 **#3** 尽管人们为改进覆盖网络做出了诸多努力，但没有一种方案能同时实现低开销、高性能、高灵活性和高兼容性。

2.4 动机：不变性

基于 §2.2 中的分析，我们进一步将额外开销的处理分为以下几类：连接跟踪、数据包过滤、主机内路由、命名空间遍历和外层头部处理 [2]。在这些开销类型中，命名空间遍历不会产生任何处理结果，而其余四类则具有不变性。

连接跟踪的不变性。据我们所知，所有常见的连接跟踪器（例如，netfilter [2], OVS [51],Cilium [15]）都具有已建立状态，这表示连接跟踪器已观察到连接的双向通信。一旦进入已建立状态，连接在完成之前就不会切换到其他状态。

数据包过滤中的不变性。 数据包过滤器可分为两类：无状态和有状态。无状态过滤器仅匹配数据包字段，通常是五元组。有状态过滤器还会考虑连接状态，该状态由连接跟踪器提供。尽管 OVS 和 netfilter 包含许多过滤器，但当流进入已建立状态后，对于具有相同五元组的数据包，允许还是拒绝的最终决定始终不变。

主机内路由的不变性。如果容器 IP 地址位于同一子网内，则只需进行二层路由，数据包即可直接转发到目标接口。否则，就需要进行三层路由，并且在转发数据包之前应修改 MAC 地址。在这两种情况下，发往同一容器 IP 地址的数据包，其主机内路由决策都是不变的。

外层头部处理的不变性。在入口路径上，检查目的地址和生存时间（TTL）等字段后，只需移除外层首部。内层首部的校验和可确保负载正确无误。出口路径会封装外层首部，其中大多数字段与路由和网络配置有关。在同一主机上，发往同一容器 IP 地址的数据包的这些字段是不变的：(1) 外层 MAC 首部存储 MAC 地址、VLAN 和以太网类型，这些内容都是不变的。(2) 外层 IP 首部中的 IP 地址、协议、生存时间（TTL）、差分服务代码点（DSCP）等都是不变的。长度、标识（ID）和校验和字段会因数据包而异，但只需少量计算即可轻松更新。(3) 外层 UDP 报头中的目的端口（VXLAN 设为 4789）和校验和（VXLAN 设为 0）[3] 是不变的 [41]。源端口只需对内层首部的五元组进行哈希计算即可得出。长度字段也可以轻松更新。(4) VXLAN 报头存储 VXLAN 网络标识符（VNI），它在覆盖网络中不会改变。

要点 **#4** 覆盖网络的额外开销呈现出不变性。这一观察促使我们引入缓存来消除额外开销。

3 ONCache

ONCache 是一种容器覆盖网络，提供基于缓存的快速路径，具有低开销、高性能、灵活性和兼容性。它本质上是 Antrea、Flannel 等标准覆盖网络的插件。ONCache 采用故障安全设计：当快速路径不可用时，流量

[2]如果启用了 Kubernetes ClusterIP 服务 [62]，那么主机间容器流量也需要进行网络地址转换（NAT），如 §3.5所述。

[3]虽然对于 GENEVE 等某些隧道协议而言，UDP 校验和是必需的，但其计算开销很低。

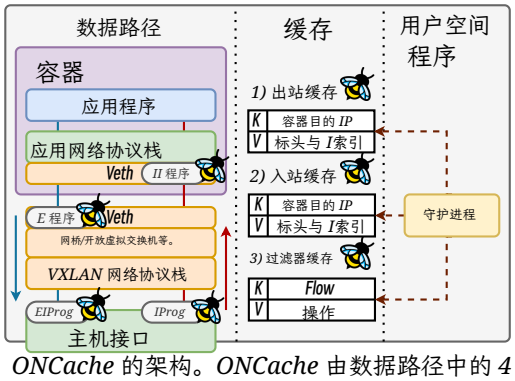

图1： *ONCache* 的架构。*ONCache* 由数据路径中的 *4* 个扩展伯克利包过滤器程序、*3* 个 *eBPF* 映射和 *1* 个用户空间程序组成。带有 *eBPF* 标志（“蜜蜂”）的组件表示扩展伯克利包过滤器程序或 *eBPF* 映射。*dIP* 是目的 *IP* 地址的缩写。

扩展伯克利包过滤器程序	缩写	挂载点
Egress-Prog	E-Prog	虚拟以太网设备的 TC 入口（主机侧）
Ingress-Prog	I-程序	主机接口的 TC 入口
Egress-Init-Prog	EI-Prog	主机接口的 TC 出口
Ingress-Init-Prog	II-Prog	虚拟以太网设备（容器侧）的 TC 入口

表 3：四个 *eBPF* 程序的挂载点。

无缝回退到标准覆盖网络。

定义缓存：受 §2.4 中讨论的不变性启发，ONCache 在每台主机上定义了三种缓存——出口缓存、入口缓存和过滤器缓存——用于存储额外开销中不变的结果。这些缓存的详细设计见 §3.1。

初始化缓存：ONCache 依靠标准覆盖网络来初始化缓存。发生缓存未命中时，ONCache 会将数据包传递给标准覆盖网络进行处理；如果该数据包所属的流达到已建立状态，则将处理结果存入缓存。详细设计见 §3.2 。

利用缓存：缓存命中时，ONCache 使用缓存的过滤结果过滤数据包，利用缓存的外层首部构造隧道数据包，然后根据缓存的路由决策将数据包重定向至目标接口。基于缓存的快速路径通过多次轻量级缓存查找，消除了 §*2.4* 中分析的多个层所需的昂贵重复处理。详细设计见 §3.3。

维护缓存：当发生网络变更事件（包括容器删除、迁移和过滤器修改）时，ONCache 会确保适时清除过时的缓存条目。详见 §3.4。

缓存兼容性：ONCache 专注于跨主机容器网络，同时兼容网络中的各种流量和功能。详见 §3.5。

可选改进： 我们提供了两项可选改进，以进一步提升性能。由于这些改进需要修改内核或隧道协议，因此不会作为默认设计采用。相关内容见§3.6。

为便于实现和部署，我们使用扩展伯克利包过滤器 [24,70]实现 ONCache。扩展伯克利包过滤器是一种技术，可在 Linux 内核等特权上下文中运行沙箱程序，从而安全、高效地扩展内核的功能。扩展伯克利包过滤器程序会附加到内核钩子点（例如流量控制（流量控制）或快速数据路径（快速数据路径） [31]），每当内核执行到这些挂载点时就会触发。扩展伯克利包过滤器会将数据存储在内核空间中的一种映射数据结构中，称为 eBPF 映射。许多先前的研究都利用扩展伯克利包过滤器实现网络功能 [4,44,56,67,72]。

如图 1 所示，ONCache 包含以下组件：(1) 四个用于数据包处理和转发的扩展伯克利包过滤器程序，其挂载点列于表 3；(2) 每台主机上三个扩展伯克利包过滤器映射，分别作为出口缓存、入口缓存和过滤器缓存；(3) 一个用户空间守护进程。与 Slim 和 Falcon 相比，ONCache 的实现简单得多。核心功能仅需 524 行扩展伯克利包过滤器 C 代码即可实现，而 Slim 则需要 2380 行代码。Falcon 使用 606 行内核代码实现，其实现和调试比扩展伯克利包过滤器更困难。核心源代码见附录 B。

我们的思路与其他基于缓存的网络技术有着根本性的不同。先前的工作，例如 OVS [53,58] 和 Andromeda [20], 利用缓存加速其数据路径上的流/规则匹配。然而，正如 §2.2 中的分析所示，即使使用了 OVS 缓存，容器覆盖网络的性能仍不理想，这表明仅缓存单个层的结果并不足够。相比之下，ONCache 是容器覆盖网络中首个跨层缓存，它将额外开销作为整体考虑，并只缓存一次结果。因此，与先前的工作相比，ONCache 能有效消除额外开销。

3.1 缓存的定义

如图 1 所示，ONCache 在每台主机上包含三个缓存：出口缓存、入口缓存和过滤器缓存。这些缓存使用扩展伯克利包过滤器的最近最少使用（LRU）哈希映射实现；当映射已满时，会逐出最近最少使用的元素，以便存储新元素。

出口缓存存储外层首部和出站主机内路由结果，这些内容对于发往同一容器 IP 地址的数据包保持不变。路由结果包括新的内层 MAC 首部（用于三层路由）和出站主机接口。为减少出口缓存的内存使用量，我们将其划分为两级： <容器目的 *IP* 地址（目的 *IP*） → 主机目的 *IP* 地址> 和 <主机目的 *IP* 地址 → 外层首部、内层 *MAC* 首部、主机接口索引>。

入口缓存存储入站主机内路由结果，这些结果对于发往同一容器 IP 地址的数据包保持不变。入口缓存的结构为 <容器目的 *IP* → 内层 *MAC* 首部、*veth*（主机侧）索引>。

filtercache 存储每个处于已建立状态的流的过滤决策。本质上，它充当流

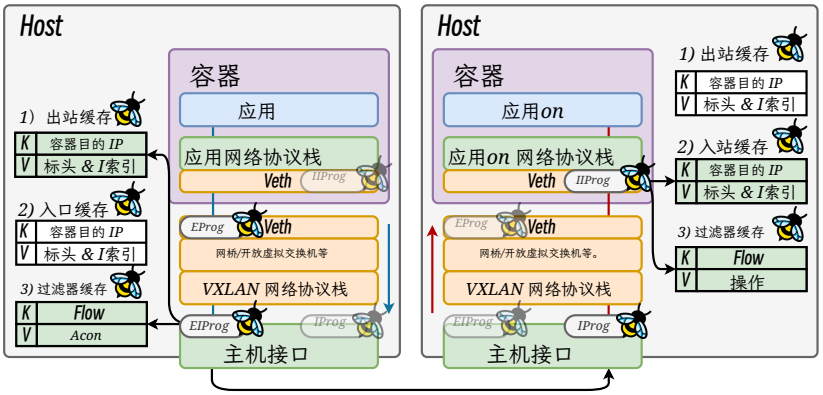

图 2：*ONCache* 的缓存初始化过程。当满足初始化要求时，*EI-Prog/II-Prog* 会更新条目（绿色阴影部分）。

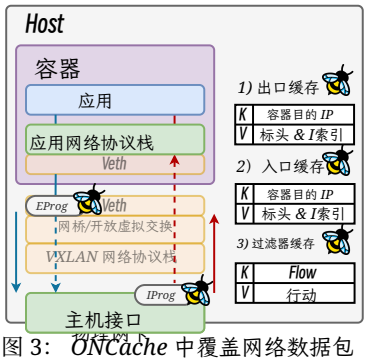

图 3：*ONCache* 中覆盖网络数据包的传送过程。跳过 *EI-Prog/II-Prog*。虚线表示重定向路径。

白名单，记录允许的流。默认情况下，一个流由五元组定义（源 IP 地址、源端口、目的 IP 地址、目的端口和传输协议）。也可以根据需要调整流的定义，例如添加 DSCP 字段以支持 DSCP 过滤器。

为了尽量减少缓存未命中，应根据集群规模和并发流数量调整缓存容量。以 Kubernetes 中最大的集群为例 [60]，每台主机最多有 110 个容器，共有 5k 台主机和 150k 个容器。每台主机最多有 1M 个并发流时，为了尽量减少缓存未命中，出站/入站/过滤器缓存每台主机分别最多占用 1.56 MB/2.2 KB/20 MB 的内存空间（详情见附录 C ）。在现代服务器中，这些内存使用量可以忽略不计。

3.2 缓存初始化

如果 ONCache 在尝试转发数据包时遇到缓存未命中，就会将该数据包交给备用覆盖网络，同时尝试初始化缓存。缓存初始化过程如图 2所示。

初始化出站路径。当出口容器数据包到达虚拟以太网设备（主机侧）时，在此处挂载的 Egress-Prog 会尝试转发该数据包。如果该数据包未命中出口缓存或过滤器缓存，Egress-Prog 就会在数据包上添加一个未命中标记。我们在内部 IP 头的 DSCP 字段中保留一位作为未命中标记。[4] Egress-Prog 随后将数据包传递给备用覆盖网络。

备用覆盖网络会继续转发、过滤并封装数据包。此外，我们将覆盖网络配置为：当数据包所属的流达到已建立状态时，为数据包添加一个已建立标记。我们使用 DSCP 字段中的另一位作为已建立标记。如附录B.2所述，上述操作只需稍微调整两条 OVS 流规则，或在 netfilter 中添加一条规则。

经过备用覆盖网络处理后，数据包到达主机接口，此时 Egress-Init-Prog 会检查是否满足以下条件以初始化缓存：(1) 数据包是隧道数据包（例如 VXLAN 数据包）；(2) 数据包带有*miss* 标记和*est* 标记。

如果满足所有要求，Egress-Init-Prog 会通过将 <容器目的 *IP* → 外层首部、内层 *MAC* 首部、主机接口索引 > 存入缓存来初始化出口缓存。ON-Cache 从 sk_buff 结构体中获取主机接口索引，并从隧道数据包中获取其他字段。此外，该流会被加入过滤器缓存的白名单。最后，数据包从主机接口发出。

初始化入口路径。 主机接口收到隧道数据包后，在此处挂载的 Ingress-Prog 会查询缓存，并尝试转发数据包。如果该数据包在入口缓存或过滤器缓存中发生缓存未命中，Ingress-Prog 就会为数据包添加 *miss* 标记，并将其传递给备用覆盖网络。

备用覆盖网络会继续处理数据包；如果其流达到已建立状态，则会为数据包添加 *est* 标记。

随后，数据包到达 veth（容器侧），Ingress-Init-Prog 会检查数据包是否同时带有 *miss* 和*est* 标记。如果是，程序就会通过将<容器目的 *IP* → 内层 *MAC* 首部> 存入入口缓存来初始化缓存，并将该流加入过滤器缓存的白名单。 <容器目的 *IP*→*veth*（主机侧）索引> 在入口缓存中的维护工作由 ONCache 守护进程在容器配置期间完成。

3.3 基于缓存的快速路径

我们介绍 ONCache 的基于缓存的快速路径（图 3）。该快速路径旨在实现故障安全，并且对应用程序和底层网络均保持透明。对于无法通过快速路径转发的数据包，ONCache 会将其传递至备用覆盖网络，而不是将其丢弃。

3.3.1 出口快速路径

出口快速路径由 Egress-Prog 实现，它按以下步骤处理出口容器数据包：

[4]我们在内部 IP 头部的 DSCP 字段中总共保留 2 位，并要求网络不要将这 2 位用于差分服务。

步骤 #1：缓存检索。Egress-Prog 首先检查数据包的流是否在过滤器缓存的白名单中，并从出口缓存中检索<容器目的 *IP* → 外层首部、内层 *MAC* 首部、主机接口索引>任何缓存未命中都会触发缓存初始化过程，如 §3.2所述。

此外，Egress-Prog 还会执行反向检查，验证容器源 IP 地址是否存在于入口缓存中，以及反向流是否也在过滤器缓存的白名单中。如果未通过反向检查，Egress-Prog 就会将数据包传递给备用覆盖网络。这是必要的，因为对于一条流，两个方向的缓存逐出以及其连接跟踪条目的过期都是异步发生的。只有在两个方向的缓存都准备就绪时才使用快速路径，这样一来，如果需要进行缓存初始化，连接跟踪就能观察到两个方向的流量，并为数据包添加 *EST*标记，从而确保初始化成功。附录D中给出了一个反例。

步骤 #2：封装与主机内路由。获取出口缓存后，Egress-Prog 接下来会重写数据包的内层 MAC 首部，并添加外层首部。需要对外层首部进行两项修改：（1）更新外层 IP 首部中的长度、ID 和校验和字段，以及外层 UDP 首部中的长度字段；（2）使用与内核相同的哈希函数计算外层 UDP 源端口。最后，Egress-Prog 调用 eBPF 辅助函数 bpf_redirect，将数据包重定向到缓存的主机接口索引。

3.3.2 入口快速路径

入口快速路径由 Ingress-Prog 实现，它通过以下步骤处理入口隧道数据包：

步骤 #1：目的地址检查。Ingress-Prog 首先验证数据包的目的 MAC 地址和 IP 地址是否与当前主机接口的地址匹配，同时检查生存时间。如果目的地址检查失败，则将数据包交由备用覆盖网络进行最终判定。

步骤 #2：缓存检索。接着，Ingress-Prog 检查数据包的流是否已列入过滤器缓存的白名单，并从入口缓存中检索<容器目的 *IP* → 内层 *MAC* 首部、*veth*（主机侧）索引>。任何缓存未命中都会触发缓存初始化过程。此外，入口路径上还会执行反向检查（见 §3.3.1）。

步骤 #3：解封装和主机内路由。Ingress-Prog 随后移除数据包的外层首部，重写内层 MAC 首部，并将数据包重定向到目标 veth（容器侧）。该重定向通过调用 eBPF 辅助函数 bpf_redirect_peer 实现。

尽管 ONCache 简化了外层首部的解封装，但并未损失任何功能。首先，负载受内层首部校验和的保护，并由容器网络协议栈进行验证。其次，分片数据包会在到达 Ingress-Prog 之前，由通用接收卸载（GRO）完成重组。此外，没有其他功能依赖外层首部。

3.4 缓存一致性

覆盖网络可能发生变化，因此 ONCache 必须在变化发生时确保缓存一致性。容器配置由缓存初始化机制处理，而其他变化则由用户空间程序 ONCache 守护进程处理。

容器删除或发生意外容器故障时，ONCache 守护进程会删除相关缓存。这可以防止具有旧 IP 地址的新容器错误地使用过时的缓存条目。

发生其他更改时，包括容器迁移或过滤器更新，ONCache 会采用 删除并重新初始化机制，通过以下四个步骤确保这些更改立即且正确地在快速路径中生效：(1) 禁用 OVS 或 netfilter 添加 *est*标记，从而暂停缓存初始化。(2) 删除受网络更改影响的缓存条目（受影响的数据包开始使用备用覆盖网络）。(3) 在备用覆盖网络中应用网络更改（网络更改立即生效）。(4) 恢复缓存初始化（重新初始化缓存条目，受影响的数据包再次开始使用快速路径）。

3.5 缓存兼容性

适用于各种流量。ONCache 旨在加速主机间容器流量，不负责处理其他类型的流量，例如主机内容器流量、容器到主机 IP 流量或容器到外部 IP 流量。这些类型的流量由备用覆盖网络处理。

ClusterIP 服务 [62] 在 Kubernetes 中，便于容器通过单个 ClusterIP 访问一组提供相同功能的容器。通常，这类服务依赖发送端主机上的 netfilter 或 IPVS（IP 虚拟服务器）[33] 进行负载均衡和目的网络地址转换（DNAT）。不过，ONCache 的快速路径会绕过 netfilter 和 IPVS。尽管如此，ONCache 仍可采用类似 Cilium 的方式支持 ClusterIP：通过 eBPF 程序和映射实现负载均衡和 DNAT [14]。此功能可集成到 Egress/Ingress-Prog 中，并与基于缓存的快速路径兼容。

支持数据平面策略。速率限制和服务质量（QoS）等数据平面策略可与 ONCache 无缝集成。在 Linux 中，这些策略通过排队规则（qdisc）实现。ONCache 的快速路径不会绕过主机接口上的 qdisc。

ONCache 不支持的过滤器。基于覆盖网络数据包哈希进行匹配的数据包过滤器 [74] 不具备任何不变性。此外，尽管 netfilter [2],OVS [51],Cilium [15]中的所有有状态过滤器都具有已建立状态，但也可能存在不具有已建立状态的有状态过滤器。

ONCache 原生不支持此类过滤器，因为不存在可以缓存的不变过滤决策。尽管如此，仍可根据具体情况使用扩展伯克利包过滤器实现此类过滤器。

与服务网格协同工作。服务网格在 Kubernetes 等容器集群中越来越受欢迎，主要用于提供安全性、可观测性和流量管理等功能 [63]。服务网格最常见的实现方式是使用称为边车代理的软件代理 [73]。边车代理是一个独立进程，与应用程序共同部署在应用网络命名空间中（例如，部署在 Kubernetes Pod 中 [34]），并且仍依赖覆盖网络进行通信。因此，ONCache 有助于边车服务网格中的通信。

与容器网络接口（CNI）的兼容性。ONCache 的当前实现基于 TC eBPF。对于数据路径中不使用 TC eBPF 的容器网络接口（例如 Antrea、Flannel 等），ONCache 可以无缝集成，无需修改任何代码。反之，对于已使用 TC eBPF 的容器网络接口（例如 Cilium、Calico），我们需要在其 eBPF 程序或内核源代码中重新实现 ONCache。

容器实时迁移。ONCache 兼容容器实时迁移 [19, 45]。通过适当的控制平面操作（参见 §3.4），可以妥善维持容器连接。相比之下，Slim 仅支持冷迁移 [74],，因为其容器连接是在主机上建立的，迁移后会失效，从而导致容器内的应用程序中断。

网络调试。ONCache 支持 ICMP，因此可以使用 ping 和 traceroute 等常见调试工具。用户还可以使用 bpftool [6] 调试 ONCache 的 eBPF 程序和映射。使用 ONCache 进行调试简单方便。相比之下，使用 Slim 调试要困难得多。Slim 不支持 ICMP，也没有用于其套接字替换机制的调试工具。

此外，ONCache 还兼容 Linux 的多种技术，例如分段卸载 [61], 校验和卸载 [10],以及扩展技术 [59]。详情请参见附录E。

3.6 可选改进

我们为快速路径提出了两项可选改进。与 ONCache 的默认设计不同，这些可选改进需要修改内核或协议。

优化重定向数据路径。重定向 和bpf_重定向_对等节点 分别在出站和入口路径上重定向数据包。如图 4 (a) 所示，这两个重定向函数是不对称的，而且出方向重定向路径未能消除遍历命名空间的开销。我们推测，对称出方向重定向（图 4 (b)）需要在容器命名空间中挂接网络 eBPF 程序，这可能会与用户应用程序冲突。尽管如此，我们仍然对优化出方向重定向数据路径的潜在收益感到好奇。

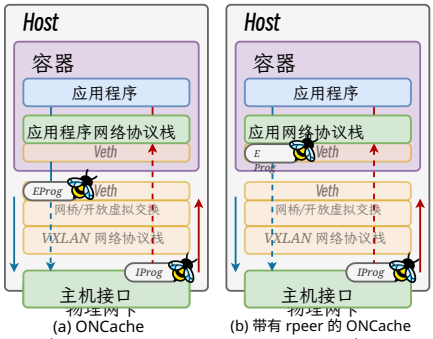

图 4：*(a)* 不含 *rpeer* 的 *ONCache*；*(b)* 含 *rpeer* 的 *ONCache* 中的快速路径。

我们在 Linux 内核中设计并实现了一种新的扩展伯克利包过滤器重定向函数，称为 bpf_重定向_rpeer。与bpf_重定向_peer相比，它提供了相反的路径：将数据包从虚拟以太网设备（容器侧）的出口端口重定向到主机接口的出口端口，如图 4 （b）所示。通过利用 bpf_重定向_rpeer，Egress-Prog 的挂载点从而变更为虚拟以太网设备（容器侧）的 TC 出口，使 ONCache 能够进一步降低出口路径上的命名空间遍历开销。

优化隧道协议。VXLAN 等传统隧道协议会因外层首部而引入传输开销，通常达到几十字节（例如 VXLAN 为 50 字节）。为消除这部分传输开销，我们提出并实现了一种基于重写的隧道协议。

借助新的隧道协议，ON- Cache 的快速路径工作如下：(1) Egress-Prog 将出口容器数据包的源/目的地 IP/MAC 地址修改为主机地址，并将一个 恢复键 写入数据包的空闲字段（例如 DSCP）。随后，数据包被重定向至底层网络。(2) Ingress-Prog 根据恢复键将所有地址恢复为其原始值，并将数据包重定向至目的地容器。恢复键与容器源/目的地 IP/MAC 地址之间的映射会在缓存初始化期间进行缓存。更多详细信息请参见附录F。

4 评估

我们通过第4.1 节中的微基准测试和第4.2节中的热门分布式应用，评估 ONCache（默认不启用可选改进）。在这两节中，启用可选改进可以进一步提升 ONCache 的性能，但需要修改内核/协议。第4.3节对启用可选改进的 ONCache 进行了评估。

我们的测试平台构建于基础 Kubernetes [35] 集群（v1.23.6）之上，包括 API 服务器、调度引擎、etcd 等。我们在 CloudLab [23] 上使用三个 c6525- 100g 节点运行实验，每个节点配备 AMD EPYC 7402P 24- 核处理器@2.80 GHz。该处理器使用超线程技术。每台机器配备 128 GB ECC 内存和一个双端口 Mellanox ConnectX-5 Ex 100 Gb 接口。我们不

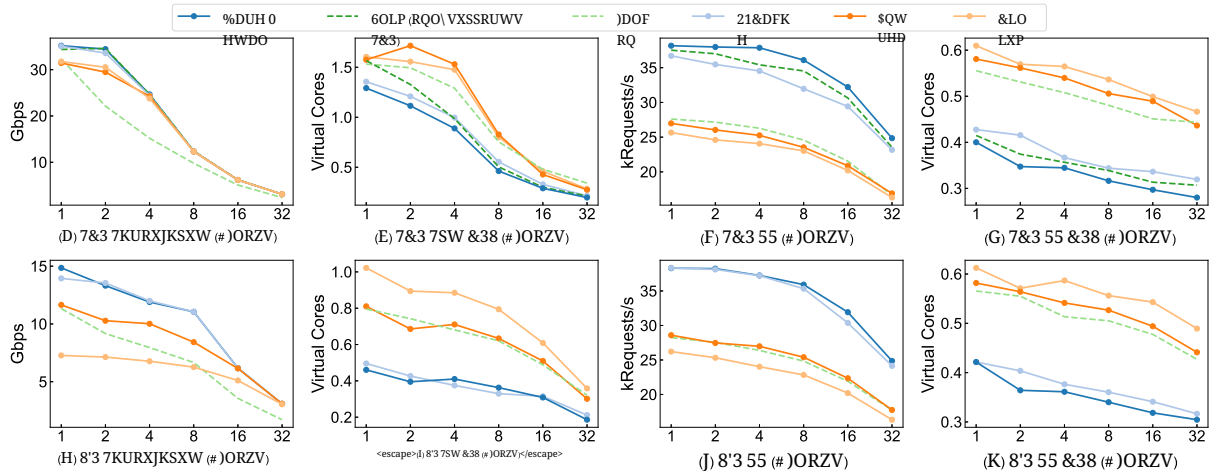

图 5：裸机、*Slim*（仅支持 *TCP*）、*Falcon*（*Linux v5.4* 内核）、*ONCache*、*Antrea* 和 *Cilium* 的 *TCP* 和 *UDP* 微基准测试结果。*Cilium* 和 *Antrea* 均提供标准覆盖网络。所有数据均为单流的平均值。在接收端主机上测量 *CPU* 利用率，并按吞吐量或 *RR* 进行归一化，再按 *Antrea* 的吞吐量或 *RR* 进行缩放。

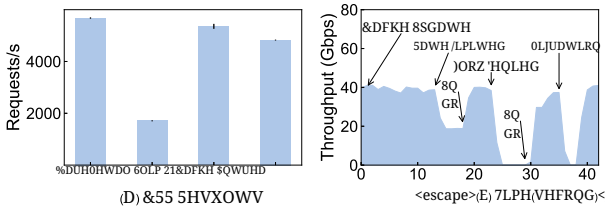

图 6：*(a)* 不同网络的连接-请求-响应（*CRR*）速率。数值越高越好。误差条表示标准差。*(b)* 功能完整性实验中的 *iperf3* 吞吐量。

更改 CloudLab 的默认分段卸载配置 [61], 校验和卸载 [10], 和扩展技术 [59]。我们使用 Ubuntu 20.04 和 5.14 版本的 Linux 内核。ON-缓存 部署为 Antrea 的插件（封装模式）。

4.1 微基准测试

4.1.1 吞吐量和延迟

我们使用 iperf3 [32] 测量吞吐量，并使用 netperf [48]测量请求-响应（RR）事务速率。RR 测试用于测量在一个连接上按顺序执行的单字节往返传输速率，其中事务速率越高，表示端到端延迟越低。在所有实验中，我们均使用 mpstat [46]测量接收端主机的 CPU 利用率。测试涵盖 TCP 和 UDP。

在所有设置中，我们都进行了并行测试。多个容器对部署在一对主机上，所有服务器容器位于一台主机上，所有客户端容器位于另一台主机上。每对容器同时开始测试，以评估 ONCache 的并行性能。

为了表示标准覆盖网络的性能（作为基线），我们选取了使用最广泛的容器网络接口：Antrea [1] 和 Cilium [13]。我们将裸机作为覆盖网络的性能上限。为了将 ONCache 与 §2.3中讨论的先前工作进行比较，我们在 Slim 上开展微基准测试[5] 和 Falcon 上开展微基准测试，分别代表套接字替换和 CPU 负载均衡的思路。

TCP 微基准测试。ONCache 在显著提升吞吐量和 RR 性能的同时，消耗的中央处理器资源更少。每流吞吐量、RR 和 CPU 利用率的结果如图所示5 （a）–（d）。

与 Antrea 相比，在 1 路和 2 路并行测试中，ONCache 的 TCP 吞吐量分别提高了 11.53% 和 13.96%。在 4、8、16 和 32 路并行测试中，所有容器网络的吞吐量都达到 100 Gb 物理网络的上限。不同并行测试中，ONCache 将每字节 CPU 利用率降低了 13.94%–34.87%。ONCache 的吞吐量和 CPU 利用率与 Slim 和裸机的结果非常接近。Falcon（仅提供 v5.4 内核中的实现）的吞吐量明显较低，原因是在我们的测试平台上，v5.4 内核的带宽本身就低于 v5.14 内核。

在所有并行测试中，ONCache 的 RR 性能比 Antrea 高出 35.81% 至 40.91%，每个 RR 的 CPU 利用率则降低了 26.02% 至 32.03%。ONCache 的 RR 结果略逊于 Slim。包括出口命名空间遍历、扩展伯克利包过滤器执行和外部报头 MTU 开销在内的开销造成了这一差距，可通过 §3.6中提出的可选改进来缓解。由于 RR 测试不会使任何 CPU 核心过载，Falcon 对 RR 结果的提升有限[6]。

UDP 微基准测试。 与 Slim 不同，ONCache 也能改善 UDP 流量，如图 5 (e) – (h) 所示。在 1、2、4、8 路并行测试中，ONCache 的 UDP 吞吐量比 Antrea 高 19.68% - 31.76%。在 16 路和 32 路并行测试中，由于带宽瓶颈，吞吐量保持不变。此外，所有测试中的单位字节 CPU 利用率降低了 29.73%

[5]Slim 不支持 UDP，因此只进行 TCP 微基准测试。[6]在我们的测试平台上，内核版本不会影响 RR 结果。

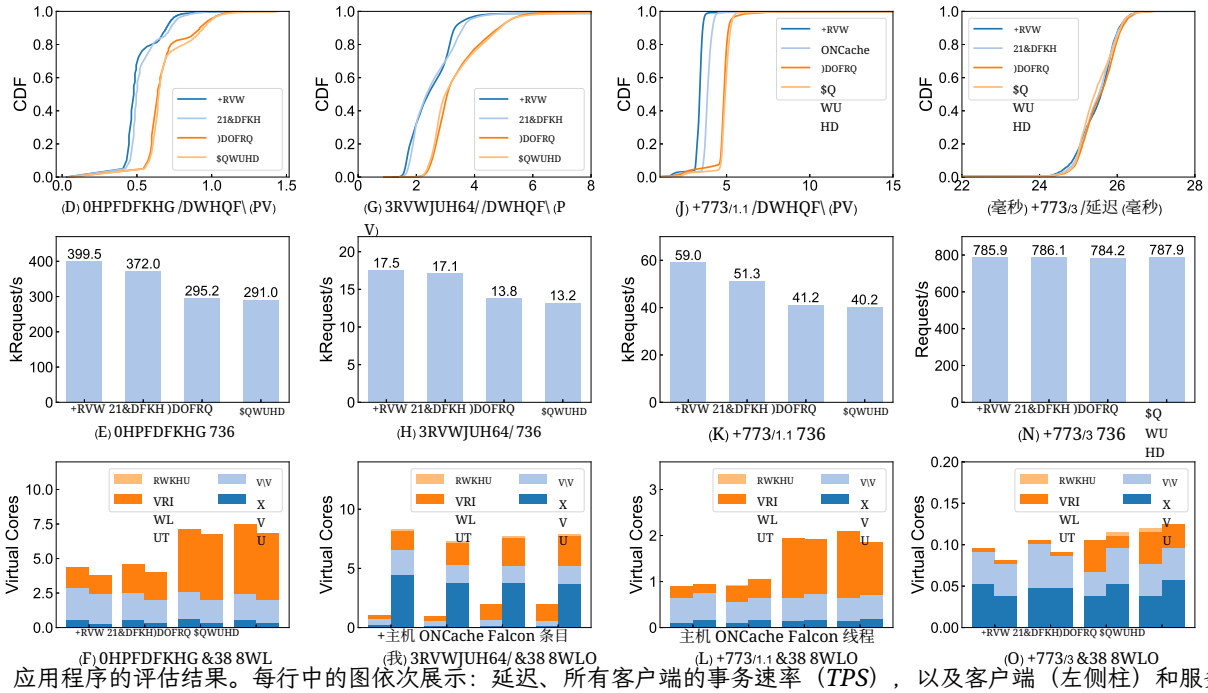

图 7： 应用程序的评估结果。每行中的图依次展示：延迟、所有客户端的事务速率（*TPS*），以及客户端（左侧柱）和服务器（右侧柱）的 *CPU* 利用率。为确保公平比较，我们按 *TPS* 对 *CPU* 利用率进行归一化，并将其缩放至 *Antrea* 的 *TPS*。*Softirq* 表示软件中断请求。

- 47.98%。ONCache 与裸机之间的吞吐量差距小于 6%。

在所有测试中，ONCache 将 UDP 请求-响应的性能提升了 34.13% - 39.12%，每次请求-响应的 CPU 利用率降低了 27.54% - 31.59%。由于与 TCP RR 实验相同的原因，其性能与 Slim 之间仍存在小幅差距。

开销量化。 我们采用 §2.2 中提出的方法量化 ONCache 的开销，并在表 2中展示结果。正如预期，ONCache 消除了所有额外开销，但出站命名空间遍历开销除外；该开销由 `bpf_redirect_rpeer` 在 §3.6中提出的方法加以解决。尽管引入了扩展伯克利包过滤器执行开销，ONCache 仍显著降低了网络协议栈的总体延迟。定量分析结果与前文给出的微基准测试结果一致。

4.1.2 缓存开销

缓存初始化。 我们使用 Netperf 的连接请求-响应（CRR）测试来测量 TCP 连接建立时间，并展示缓存初始化开销。在 CRR 中，每次 RR 事务都需要建立新的 TCP 连接，这会带来额外的缓存初始化开销。

CRR 结果如图6 （a）所示。ONCache 优于 Antrea，但不及裸机。这是因为在缓存初始化之前，ONCache 依赖 Antrea 处理前 3 个数据包。在这部分中，ONCache 的表现与 Antrea 相同。之后，1 字节数据的往返（请求-响应部分）通过快速路径完成，其表现优于 Antrea。与 Antrea 相比，Falcon 无法从 CRR 中受益，原因是与 TCP RR 中的原因相同。Slim 的表现明显较差。原因是 Slim 需要先为服务发现建立覆盖网络连接，这会带来额外的几次 RTT。

缓存干扰。 我们在持续更新缓存条目的同时运行 iperf3 测试，以展示潜在的缓存干扰开销。在此实验中，所有缓存容量均设为 512，缓存替换策略为 LRU。我们使用脚本向出口缓存持续插入 1000 个冗余缓存条目，随后将其删除，共进行 2 轮。测试耗时约 8 秒。如图6 （b）所示，在前 8 秒内，iperf3 流的吞吐量没有明显波动。

缓存可扩展性。 我们使用包含 150,000 个条目的满载出口缓存进行 TCP RR 测试（对应 §3.1 中提到的最大 Kubernetes 集群）。正如预期，RR 性能不受影响，体现了哈希映射固有的可扩展性。

4.1.3 功能完整性

数据平面策略。我们测试数据平面策略（例如速率限制）是否能在 ONCache 中正常工作。如图6 (b) 所示，在没有速率限制的情况下，吞吐量约为 39 Gbps。随后，我们在主机接口上将速率限制为 20 Gbps，吞吐量降至约 18.5Gbps。移除速率限制后，吞吐量恢复到原来的水平。

数据包过滤器。我们测试数据包过滤器能否在 ONCache 中正常工作。在使用 ONCache 进行 iperf3 测试期间，我们应用

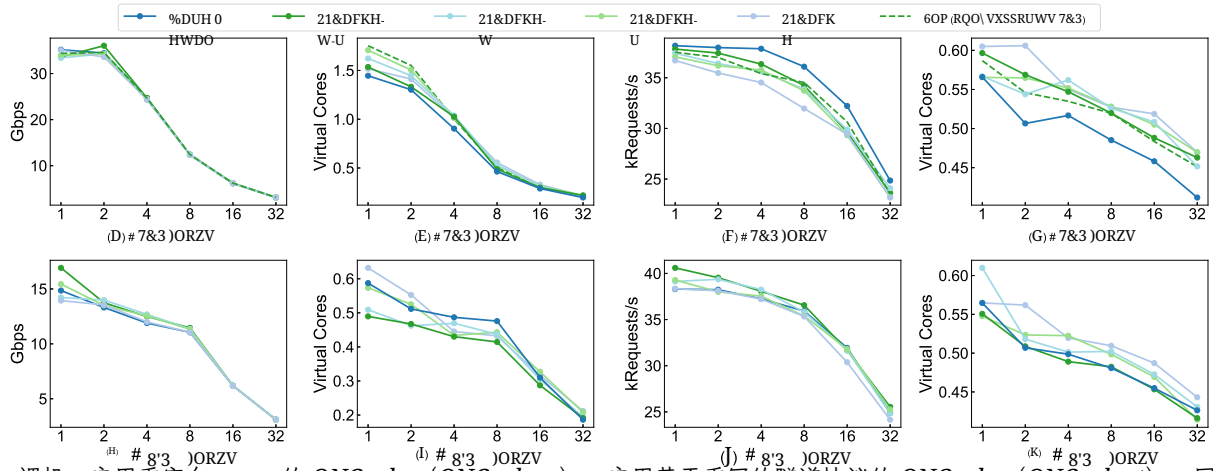

图 8： 裸机、启用重定向 *rpeer* 的 *ONCache*（*ONCache-r*）、启用基于重写的隧道协议的 *ONCache*（*ONCache-t*）、同时启用两者的 *ONCache*（*ONCache-t-r*）以及两者均未启用的 *ONCache*（*ONCache*）的 *TCP* 和 *UDP* 微基准测试结果。*CPU* 利用率按吞吐量或 *RR* 进行归一化，并以裸机的吞吐量或 *RR* 为基准进行缩放。

一个简单的数据包过滤器，使用 §3.4 中提出的删除并重新初始化机制来拒绝此 iperf3 流。正如预期，吞吐量降至 0，如图6 (b) 所示。移除过滤器后，吞吐量恢复。

容器实时迁移。我们评估了 ONCache 是否能在容器实时迁移期间正常工作。由于 Kubernetes 本身不支持热迁移，我们通过在容器保持运行的同时修改主机 IP 地址和 VXLAN 隧道来模拟热迁移。如图 6 (b) 所示，迁移开始时（主机 IP 地址发生变化），吞吐量降至 0；迁移完成后（VXLAN 隧道更新），吞吐量在约 2 秒后恢复。

数据包过滤器和容器实时迁移实验都会引起网络变化，因此需要使用删除并重新初始化机制（在 §3.4 中提出）来应用这些变化。因此，这些实验也展示了删除并重新初始化机制的有效性。

4.2 应用程序

我们在三个真实世界的应用程序上评估了 ONCache：内存键值存储 Memcached [42],数据库 Post- greSQL [55],和 Web服务器 Nginx [50]。我们以 Antrea 作为性能基线，以 Docker主机网络 [21]作为上限。我们还评估了 Falcon。由于 Slim 的性能与主机网络相近，因此本节不作介绍。值得注意的是，Slim 仅支持 TCP，这限制了其兼容性。

Memcached。我们在两台主机上各部署一个容器。一个容器运行 Memcached [42] 服务器（v1.6），另一个则作为客户端运行标准的 Memcached基准测试工具 memtier [43]（v2.0.0）。实验开始时，客户端启动 4 个线程，每个线程建立 50 个连接。SET 和 GET请求的比例为 1:10。memtier 以尽可能快的速度运行。实验期间，我们使用 mpstat 记录两台主机上的 CPU 利用率。

ONCache 的延迟低得多，事务速率也更高，优于 Antrea 的操作速率。图 7 （a）展示了 GET请求延迟的 CDF图。平均延迟降低了 22.71%，而 99.9 百分位延迟降低了 27.69%。与主机网络相比，延迟差距在 6% 以内。事务速率如图7 （b）所示。ONCache 的表现比 Antrea 高出 27.83%，但与主机网络相比仍有 7% 的差距。

如图所示，ONCache 还显著降低了每笔事务的 CPU 利用率 7 (c)。ONCache 在客户端和服务器上的每笔事务 CPU 利用率分别降低了 38.91% 和 40.98%。但其利用率仍比主机网络高 4.09% 和 6.62%。

PostgreSQL。我们在不同主机上的一对容器中部署 PostgreSQL数据库 [55]（v15.3）及其基准测试工具 pgbench [54]。Pgbench 实现 TPC-B基准测试，创建一个包含 500 万个银行账户的数据库，并发地执行 50 个客户端。Pgbench 以尽可能快的速度运行。我们通过 mpstat 测量 CPU 利用率。

ONCache 有利于降低 PostgreSQL 的延迟并提高事务速率，如图 7 （d）和（e）所示。与 Antrea 相比，ONCache 将平均延迟降低了 22.34%，将事务速率降低了 29.40%，同时与主机网络相比，二者的差距分别为 2.30% 和 2.54%。

如图所示，ONCache 还降低了 CPU 利用率7 （f）。客户端和服务器的每笔事务 CPU 利用率分别降低了 51.84% 和 7.26%。服务器的降幅较小，这是因为其用户态 CPU 利用率相对较高。ONCache 使服务器的软中断利用率降低了 27.36%。

Nginx。为评估 HTTP 服务器的性能，我们针对 HTTP/1.1 和 HTTP/3（QUIC）进行负载测试， [57],分别代表基于 TCP 和基于 UDP 的 HTTP。HTTP/1.1 禁用 SSL，而 HTTP/3 启用 SSL。我们使用 Nginx [50]（v1.25.1）和 h2load [30]（v1.55.1）对 HTTP 服务器进行基准测试。对于 HTTP/1.1，h2load 运行 100 个客户端；对于 HTTP/3，则运行 10 个客户端。每个客户端发起 2 个并发流

并向服务器请求一个 1 KB 文件。h2load 以尽可能快的速度运行。

如图所示，ONCache 提升了 HTTP/1.1 的延迟和事务速率7 (g) 和 (h)。ONCache 将 HTTP/1.1 请求延迟降低了 21.53%，并将事务速率提高了 27.43%。ONCache 与主机网络之间仍存在性能差距（延迟方面为 15.04%，事务速率方面为 13.12%）。造成这一差距的原因与 TCP RR 中分析的相同，因为事务速率足够高，足以显现开销。对于 HTTP/3，性能明显较差，且在不同网络中保持一致，如图7 (j) 和 (k) 所示。这可能归因于 Nginx 的实验性 QUIC 支持 [26]。

如图所示，ONCache 降低了 HTTP/1.1 和 HTTP/3 的每笔事务的 CPU 利用率7 (i) 和 (l)。对于 HTTP/1.1，客户端的 CPU 利用率降低了 56.26%，服务器的 CPU 利用率降低了 43.52%。对于 HTTP/3，客户端和服务器的 CPU 利用率分别降低了 11.80% 和 26.76%。

4.3 可选改进的评估

我们采用第4.1节所述的方法，对带有重定向 rpeer（ONCache-r）、基于重写的隧道协议（ONCache-t）以及同时采用这两项改进（ONCache-t-r）的 ONCache 进行了评估。结果如图8所示。本节中的基线是未采用可选改进的 ONCache。

如图8 （c）（g）所示，可选改进有助于提高 RR 事务速率。ONCache-t、ONCache-r 和 ONCache-t-r 分别将单并行 TCP RR 性能提高了 1.96% 、 0.97% 和 3.08% （对于 UDP，则分别提高了 2.04%、2.43% 和 5.87%）。在多并行测试中，可选改进同样有助于提升 RR 性能。ONCache-t-r 带来的 RR 性能提升最大，几乎等于 ONCache-t 和 ONCache-r 各自提升幅度之和，并且达到了几乎与 Slim 相同的 RR 性能。

这些可选改进通常会降低 CPU 利用率，如图 8 （b）（d）（f）（h）所示。由于系统中其他进程产生的噪声，CPU 利用率的降幅在不同实验之间有所差异。

我们还使用 §4.2 中所用的应用评估了这些可选改进。结果见附录 G。

5 讨论

为什么使用 TC hook? 用于网络的 eBPF hook 主要有两种类型：快速数据路径和 TC [5]。与快速数据路径相比，TC eBPF 程序不需要驱动程序支持，能够以更低的开销重定向数据包，与 Linux 流量控制（tc）模块兼容，并且可以挂载到入站和出站端口。因此，TC 是 ONCache 的更好选择。

基于 eBPF 的 ONCache 的安全性。 只有以特权用户（root 用户）身份运行或拥有 CAP_BPF 能力的进程才能加载 eBPF 程序并读取/写入 eBPF 映射，除非特权用户启用非特权 eBPF [25][7]。因此，ONCache 的扩展伯克利包过滤器组件受到扩展伯克利包过滤器权限控制的保护。此外，ONCache 的快速路径位于内核模式中，应用无法访问。相比之下，Slim 存在严重的安全性问题，因为它将主机命名空间文件描述符暴露给容器，从而破坏了 Linux 命名空间提供的资源隔离。Slim 的内核模块可以缓解这些安全性问题，但资源隔离仍会受到削弱。

6 相关工作

网络虚拟化中的缓存。 OVS [53, 58] 使用缓存来加速流匹配。然而，正如 §2.2 中所分析的，覆盖网络中仍存在显著的额外开销。Andromeda [20], 这一虚拟机集群网络虚拟化协议栈维护着一个路由与过滤决策缓存，其理念与 OVS 类似。尽管如此，已有工作都没有像 ONCache 那样量化容器覆盖网络的性能，并采用跨层缓存来弥合容器覆盖网络与裸机之间的性能差距。

基于 eBPF 的容器网络。 ONCache 与现有的基于 eBPF 的容器网络（如 Cilium）有显著不同 [13] 和 Calico [7]。例如，在 Cilium 中，基于 eBPF 的数据路径旨在用 eBPF 程序替换主机网络协议栈中的 netfilter，并且在容器数据包直接转发到底层网络时效果显著 [17]。然而，正如第 2.2 节所分析的，Cilium 未能改善覆盖网络，额外开销依然存在。相比之下，如表 2 所示，ONCache 旨在通过提出的缓存消除额外开销。

服务网格边车代理优化。 虽然边车代理简化了流量管理、策略执行、网络安全等，但也带来了显著开销。Zhu 等人 [73] 提出了 MeshInsight，用于量化服务网格边车代理的开销；而 SPRIGHT [56],和 mRPC [11] 则通过不同机制优化边车代理。然而，这些工作关注的是优化整体的主机内服务网格架构，而非主机间容器通信。

7 结论

我们设计并实现了 ONCache，这是一种基于缓存的容器覆盖网络，能够在不牺牲灵活性和兼容性的前提下，有效缩小裸机与覆盖网络之间的性能差距。在微基准测试和应用实验中，ONCache 在吞吐量和延迟方面均显著优于标准覆盖网络，同时降低了 CPU 利用率。ONCache 可与 Antrea、Flannel 等现有容器网络接口无缝集成。

[7]尽管存在完全不需要该权限的用户空间 eBPF 实现 [68]，但它们是在用户空间中执行 eBPF 程序，与 ONCache 采用的 eBPF 不同。

参考文献

[1] Antrea。https://antrea.io/。

[2] P. Ayuso。netfilter 的连接跟踪系统。*Login:The Usenix Magazine*，31:34–39，2006。

[3] BCC。https://github.com/iovisor/bcc。

[4] Matteo Bertrone、Sebastiano Miano、Fulvio Risso 和 Massimo Tumolo。使用扩展伯克利包过滤器和 iptables 加速 Linux 安全。在*ACM SIGCOMM 2018* 海报与演示会议论文集，第 108–110 页，2018 年。

[5] BPF 和 XDP 参考资料。https://docs.cilium.io/en/stable/bpf/。[6] bpftool。https://github.com/libbpf/bpftool。[7]Calico。https://www.tigera.io/project-calico/。[8] Emiliano Casalicchio。容器编排：一项综述。*Systems Modeling: Methodologies and Tools*，第 221–235 页，2019 年。[9] Emiliano Casalicchio 和 Stefano Iannucci。容器技术的最新进展：应用程序、编排与安全性。*Concurrency and Computation:PracticeandExperience*，32(17):e5668，2020 年。[10]校验和卸载。https://docs.kernel.org/networking/checksum-offloads.html。[11] Jingrong Chen、Yongji Wu、Shihan Lin、Yechen Xu、Xinhao Kong、Thomas Anderson、Matthew Lentz、Xiaowei Yang 和 Danyang Zhuo。将远程过程调用作为托管系统服务。在第 *20* 届 *USENIX* 网络系统设计与实现研讨会（*NSDI 23*）论文集，第 141–159 页，2023 年。[12] Sunyanan Choochotkaew、Tatsuhiro Chiba、Scott Trent 和 Marcelo Amaral。通过透明的 BPF 驱动套接字替换绕过容器覆盖网络。在*2022* 年 *IEEE* 第 *15* 届云计算国际会议（*CLOUD*）论文集，第 134–143 页。IEEE，2022 年。

<escape>[13] </escape>Cilium。https://cilium.io/。[14]Cilium 服务负载均衡。https://docs.cilium.io/en/stable/network/kubernetes/intro/#service-load-balancing。

[15]cilium/bpf/lib/连接跟踪.h。 https://github.com/cilium/cilium/blob/main/bpf/lib/连接跟踪. h。[16] Cilium - 扩展伯克利包过滤器数据路径介绍。https://docs.cilium.io/en/stable/network/ebpf/intro/。

[17] CNI 基准测试：了解 Cilium 网络性能。https://cilium.io/blog/2021/05/11/ cni-benchmark. [18]状态机。https://rlworkman.net/howtos/iptables/chunkyhtml/c1276.html.[19] CRIU。https://criu.org/Main_页面。[20] Michael Dalton、David Schultz、Jacob Adriaens、Ahsan Arefin、Anshuman Gupta、Brian Fahs、Dima Rubinstein、Enrique Cauich Zermeno、Erik Rubow、James Alexander Docauer 等。Andromeda：云网络虚拟化中的大规模性能、隔离与敏捷性。载于第 15 届 USENIX 网络系统设计与实现研讨会（NSDI 18），第 373–387 页，2018 年。[21] Docker。https://www.docker.com/.[22]Docker Swarm。https://docs.docker.com/get-started/orchestration/.[23] Dmitry Duplyakin、Robert Ricci、Aleksander Maricq、Gary Wong、Jonathon Duerig、Eric Eide、Leigh Stoller、Mike Hibler、David Johnson、Kirk Webb 等。CloudLab 的设计与运行。载于 2019 年 USENIX 年度技术会议（USENIX ATC 19），第 1–14 页，2019 年。[24] 扩展伯克利包过滤器。https://ebpf.io/.[25]扩展伯克利包过滤器安全性。https://ebpf.io/what-is-ebpf/ #ebpf-safety.[26] 模块 ngx_http_v3_module。https://nginx.org/ en/docs/http/ngx_http_v3_module.html.[27]brendangregg/FlameGraph。https://github.com/ brendangregg/FlameGraph.[28] Flannel。https://github.com/flannel-io/ flannel. [29] Jesse Gross、Ilango Ganga 和 T. Sridhar。GENEVE：通用网络虚拟化封装。RFC 8926，2020 年 11 月。[30]h2load。https://nghttp2.org/documentation/h2load-howto.html.[31] Toke Høiland-Jørgensen、Jesper Dangaard Brouer、Daniel Borkmann、John Fastabend、Tom Herbert、David Ahern 和 David Miller。快速数据通路：操作系统内核中的快速可编程数据包处理。载于第 14 届新兴网络实验与技术国际会议论文集，第 54–66 页，2018 年。

[32] iperf. https://github.com/esnet/iperf.

[33]IPVS。
http://www.linuxvirtualserver.org/software/ipvs.html。

[34]边车容器。
https://kubernetes.io/docs/concepts/workloads/pods/sidecar-containers/。

[35] Kubernetes。https://kubernetes.io/。

[36] Jiaxin Lei、Manish Munikar、Hui Lu 和 Rao Jia。通过数据包级并行加速容器覆盖网络中的数据包处理。载于 *2023 IEEE*国际并行与分布式处理研讨会（*IPDPS*），第79–89页。IEEE，2023年。

[37] Jiaxin Lei、Manish Munikar、Kun Suo、Hui Lu 和 Jia Rao。容器覆盖网络中的数据包处理并行化。*EuroSys2021*，2021 年。[38] Jiaxin Lei 、 Kun Suo 、Hui Lu 和 Jia Rao。应对容器覆盖网络内核网络协议栈的并行化挑战。载于 第*11*届*USENIX*云计算热点主题研讨会（*HotCloud 19*），2019年。[39] Fusheng Lin、Xin Zhang、Guo Chen、Li Chen、Kenli Li 和 Hongbo Jiang。Slim and fast：采用快速连接建立的低开销容器覆盖网络。*IEEE*云计算汇刊，2023年。[40] Hao Liu、Wenxin Li、Yiren Pang、Renjie Pei、Yitao Hu、Yuan Liu、Lide Suo 和 Keqiu Li。加速容器覆盖网络中延迟敏感型应用的数据传输。 *IEEE*并行与分布式系统汇刊，2023年。[41] Mallik Mahalingam、Dinesh Dutt、Kenneth Duda、Puneet Agarwal、Larry Kreeger、T. Sridhar、Mike Bursell 和 Chris Wright。虚拟可扩展局域网（VXLAN）：在三层网络上构建虚拟化二层网络覆盖的框架。RFC 7348，2014年8月。[42] memcached。https://memcached.org/。[43]memtier_基准测试。https://github.com/RedisLabs/memtier_基准测试。[44] Sebastiano Miano、Fulvio Risso、Mauricio Vásquez Bernal、Matteo Bertrone 和 Yunsong Lu。微服务时代基于扩展伯克利包过滤器的网络功能框架。*IEEE* 网络与服务管理汇刊，18(1)：133–151，2021年。[45] Andrey Mirkin、Alexey Kuznetsov 和 Kir Kolyshkin。容器检查点与热迁移。载于 *Linux*研讨会论文集，第2卷，第85–90页，2008年。[46] Mpstat · https://linux · die · net/man/1/mpstat ·

[47] netfilter。https://www.netfilter.org/。

[48]Netperf。
https://hewlettpackard.github.io/netperf/。

[49] netpipe(1) - Linux 手册页。
https://linux.die.net/man/1/netpipe。

[50] Nginx。https://nginx.org/。

[51] OVS 连接跟踪教程。
https://docs.openvswitch.org/en/latest/tutorials/ovs-conntrack/。

<escape>[52] </escape>perf：使用性能计数器进行 Linux 性能分析。
https://perf.wiki.kernel.org/index.php/Main<escape>_</escape>页面。

[53] Ben Pfaff、Justin Pettit、Teemu Koponen、Ethan Jackson、Andy Zhou、Jarno Rajahalme、Jesse Gross、Alex Wang、Joe Stringer、Pravin Shelar 等人。《Open vSwitch 的设计与实现》。发表于 第 *12* 届 *USENIX* 网络系统设计与实现研讨会（*NSDI 15*），第 117–130 页，2015 年。[54]pgbench。
https://www.postgresql.org/docs/current/pgbench.html。[55] PostgreSQL。https://www.postgresql.org/。[56] Shixiong Qi、Leslie Monis、Ziteng Zeng、Ian-chin Wang 和 K. K. Ramakrishnan。《Spright：从无服务器计算中提取服务器！基于扩展伯克利包过滤器的高性能事件驱动共享内存处理》。发表于 *ACM SIGCOMM 2022* 会议论文集，第 780–794 页，2022 年。[57] QUIC。https://www.chromium.org/quic/。[58] Alon Rashelbach、Ori Rottenstreich 和 Mark Silberstein。《通过计算缓存扩展 Open vSwitch》。发表于第 *19* 届 *USENIX* 网络系统设计与实现研讨会（*NSDI 22*），第 1359–1374 页，2022 年。[59] Linux 网络协议栈中的扩展。
https://docs.kernel.org/networking/scaling.html。[60]大型集群的注意事项。
h
t
t
p
s://kubernetes.io/docs/setup/best-practices/cluster-large/。
[61] 分段卸载。
h
t
t
ps://docs.kernel.org/networking/segmentation-offloads.html。
[62] 服务。
h
t
t
p
s://
kubernetes.io/docs/concepts/services-networking/service/。

[63] 服务网格日益兴起，但我们仍需加深理解并积累实践经验。
https://www.cncf.io/wp-content/uploads/2022/05/CNCF_服务_网格_微型调查_最终版.pdf。

[64] 单根 I/O 虚拟化。https://pcisig.com/specifications/iov/single_root/.[65] Kun Suo、Yong Zhao、Wei Chen 和 Jia Rao。容器网络分析与实证研究。载于 IEEE INFOCOM 2018—IEEE 通信会议论文集，第 189–197 页。IEEE，2018 年。[66] Kubernetes 网络模型。https://kubernetes.io/docs/concepts/services-networking/.[67] William Tu、Yi-Hung Wei、Gianni Antichi 和 Ben Pfaff。十年后重新审视 Open vSwitch 数据平面。载于 ACM SIGCOMM 2021 会议论文集，第 245–257 页，2021 年。[68] iovisor/ubpf。https://github.com/iovisor/ubpf.[69]虚拟以太网设备。https://man7.org/linux/man-pages/man4/veth.4.html.[70] Marcos A. M. Vieira、Matheus S. Castanho、Racyus D. G. Pacífico、Elerson R. S. Santos、Eduardo P. M. Câmara Júnior 和 Luiz F. M. Vieira。使用扩展伯克利包过滤器和快速数据路径实现快速数据包处理：概念、代码、挑战与应用。ACM计算综述（CSUR），53(1)：1–36，2020 年。[71]使用扩展伯克利包过滤器优化虚拟以太网设备。https://cilium.io/blog/2020/11/10/ cilium-19/#veth.[72] Yang Zhou、Zezhou Wang、Sowmya Dharanipragada 和 Minlan Yu。Electrode：利用扩展伯克利包过滤器加速分布式协议。载于第 20 届 USENIX 网络系统设计与实现研讨会（NSDI 23），第 1391–1407 页，2023 年。[73] Xiangfeng Zhu、Guozhen She、Bowen Xue、Yu Zhang、Yongsu Zhang、Xuan Kelvin Zou、XiongChun Duan、Peng He、Arvind Krishnamurthy、Matthew Lentz 等。剖析服务网格边车的开销。载于 2023 年 ACM 云计算研讨会论文集，第 142–157 页，2023 年。[74]Danyang Zhuo、Kaiyuan Zhang、Yibo Zhu、Hongqiang Harry Liu、Matthew Rockett、Arvind Krishnamurthy 和 Thomas Anderson。Slim：操作系统内核对低开销容器覆盖网络的支持。载于第 16 届 USENIX 网络系统设计与实现研讨会（NSDI 19），第 331–344 页，2019 年。

定量分析网络协议栈

在本节中，我们介绍如何对网络协议栈进行定量分析。我们分析使用 NPtcp 执行 1 字节 TCP RR 测试期间产生的开销 [49]。

首先，我们分两个步骤对不同数据路径中的开销进行定性分析：(1) 使用 perf 工具 [52],对网络协议栈函数进行为期 10 秒的分析，随后使用火焰图生成器 [27] 生成火焰图。(2) 根据火焰图，我们将网络协议栈函数归类到不同的开销部分中，例如应用程序网络协议栈、OVS 等。

接下来，我们使用扩展伯克利包过滤器逐一测量网络协议栈函数的执行时间。由于网络协议栈函数可能同时在应用程序网络协议栈和 VXLAN 网络协议栈中执行，因此需要通过调用栈对它们进行区分。我们借助 BPF 编译器集合（BCC） [3] 工具实现了两个扩展伯克利包过滤器程序。这些程序挂载到内核探针 和 内核返回探针 挂载点，分别在每个特定函数的入口和返回点执行。它们捕获当前调用栈及时间戳，并将函数（及其调用栈）和执行时间存储在 eBPF 映射中。对于每个函数，我们记录一秒内的所有执行时间样本，并计算这些样本的平均值，将其作为最终结果，如表 2所示。

B ONCache 的核心代码

本节介绍 ONCache 的 eBPF 程序核心代码，以帮助理解 ONCache 的设计。

B.1 本地缓存定义

本节中的代码实现了 §3.1 中所述的功能。

出口缓存：

```
1 结构 出口信息 { 2 unsignedchar outer_header[64];
3     __u32 接口索引; 4 }; 5 结构 bpf_elf_映射 SEC("映射") egressip_缓存 ={ 6 .type =BPF_MAP_TYPE_LRU_HASH, 7 .size_键 =大小(__be32), 8 .size_值 =大小(__be32), 9 .pinning =PIN_GLOBAL_NS, 10 .max elem = 4096,11 }; 12 结构 bpf_elf_映射 SEC("映射") 出站_缓存 ={ 13 .type =BPF_MAP_TYPE_LRU_HASH, 14 .size_键 =大小(__be32), 15 .size_值 =大小(结构 出口信息), 16 .pinning =PIN_GLOBAL_NS, 17 .max_elem = 1024, 18 };
```

入口缓存。

```
结构体入站信息 {
    __u32 接口索引;
    无符号字符型 目的MAC地址[ETH_ALEN];
    无符号字符型 smac[ETH_ALEN];
};
结构体bpf_elf_映射 SEC("映射") 入站_缓存 ={
    .type = BPF_MAP_TYPE_LRU_HASH,
    .大小_键 = 大小(__be32),
    .大小_值 = 大小(结构体 入站信息),
    .固定 = 固定_全局_命名空间,
    .max_元素 = 1024,
};
```

过滤器缓存。

```
结构体操作 {
    __u16 入站;
    __u16 出站;
};
结构体bpf_elf_映射 SEC("maps") 过滤器_缓存 ={
    .type = BPF_MAP_TYPE_LRU_HASH,
    .大小_键 = 大小(结构体 五元组),
    .大小_值 = 大小(结构体 操作),
    .固定 = 固定_全局_命名空间,
    .最大_元素 = 4096,
};
```

B.2 缓存初始化

本节中的代码实现了§3.2中所描述的功能。

初始化出站路径。TC_ACT_OK 表示内核应继续执行常规数据包处理。ONCache 使用 TC_ACT_OK 将数据包传递至备用覆盖网络。

```
1 // 检查是否未命中且带有 est 标记。 2 if ((inner_iph->tos & 0xc) != 0xc) return TC_ACT_OK; 3 // 更新过滤器缓存 4 struct 五元组 tuple_; 5 if (parse_5tuple_e(inner_iph, data_end, &tuple_)) return TC_ACT_OK; 6 struct 操作 eaction_ ={ 7 .出站 = 1, 8 .入站 = 09 }; 10 if(bpf_map_update_elem(&过滤器缓存_, &tuple_, &eaction_, BPF_NOEXIST)) { 11 struct 操作* action_ =bpf_map_lookup_elem(&过滤器缓存_, &tuple_); 12 if (action_) action_->出站 = 1; 13 } 14 // 更新出口缓存 15 struct 出口信息 egressinfo_; 16 initegressinfo(&egressinfo_, skb); 17 if(bpf_map_update_elem(&出口缓存_, &outer_iph->daddr, &egressinfo_, BPF_NOEXIST)) 18 return TC_ACT_OK; 19 if(bpf_map_update_elem(&egressip_cache, &inner_iph->daddr, &outer_iph->daddr, BPF_NOEXIST)) 20 return TC_ACT_OK; 21 // 清除 TOS 标记。 22 set_ip_tos(skb, 50, 0);
```

```
// Drop packets reported as invalid by conntrack.
table=ConntrackState, priority=210,ct_state=+inv+trk,ip actions=drop
// Forwarded tracked Service packets
table=ConntrackState, priority=200,ct_state=-new+trk,ct_mark=0x10/0x10,ip\
    actions=load:0x1->NXM_OF_IP_TOS[3],load:0x1->NXM_NX_REG0[9],\
    resubmit(,AntreaPolicyEgressRule)
// Forwarded tracked non-Service packets
table=ConntrackState, priority=190,ct_state=-new+trk,ip \
    actions=load:0x1->NXM_OF_IP_TOS[3],resubmit(,AntreaPolicyEgressRule)
// Match the first packet of untracked connection and forward it
table=ConntrackState, priority=0 actions=resubmit(,PreRoutingClassifier)
```

图9： *Antrea* 中 *OVS* 流的组成部分。我们引入的操作以红色字体突出显示；如果流达到已建立状态，我们会通过此操作将预定义的 *DSCP* 位设为 *1*。

初始化入口路径。

1 // 检查是否未命中且已标记 est。 2 if ((iphdr->tos& 0xc) != 0xc) return TC_ACT_OK; 3 // 更新入站缓存 4 struct ingressinfo* ingressinfo_ =bpf_map_lookup_elem(&ingress_cache, &iphdr->daddr); 5 if (!ingressinfo_) { 6 return TC_ACT_OK; 7 } else {8 __builtin_memcpy(ingressinfo_->dmac, eth->h_dest, ETH_ALEN);9 __builtin_memcpy(ingressinfo_->smac, eth->h_source, ETH_ALEN); 10 } 11 // 更新过滤器缓存 12 struct fivetupletuple_; 13 if (parse_5tuple_in(iphdr, data_end, &tuple_)) return TC_ACT_OK; 14 struct action eaction_ ={ 15 .egress = 0, 16 .ingress = 117 }; 18 if(bpf_map_update_elem(&filter_cache, &tuple_, &eaction_, BPF_NOEXIST)) { 19 struct action* action =_bpf_map_lookup_elem(&filter_cache, &tuple_); 20 if (action_) action_->ingress = 1; 21 } 22 // 清除 TOS 标记。 23 set_ip_tos(skb, 0, 0);

在 OVS 或 netfilter 上添加额外配置。可以通过 OVS 或 netfilter 添加 *est* 标记。

对于 OVS，此操作需要修改两个 OVS 流，如图9所示。原本，这两个流用于转发非新建状态的连接跟踪数据包。我们利用它们在转发数据包的同时添加 *est* 标记。我们引入的操作以红色字体标出：当流到达已建立状态时，将预先指定的 DSCP 位设为 1。

或者，也可以通过在 netfilter 中添加一条新规则来独立实现此操作，具体如下。

```
iptables -t mangle -A FORWARD -m 连接跟踪 --ctstate 已建立 -m dscp --dscp 0x1 -j DSCP --set-dscp 0x3
```

B.3 基于缓存的快速路径

本节中的代码实现了§3.3中所述的功能。

B.3.1 出站数据路径

步骤 #1：缓存检索。

1 struct fivetuple tuple_; 2 if (parse_5tuple_e(iphdr, data_end, &tuple_)) return TC_ACT_OK; 3 struct action *action =_bpf_map_lookup_elem(&filter_cache, &tuple_); 4 if (!action_ ||!(action_->入站 & action_->出站)) { 5 set_ip_tos(skb, 0, 0x4); 6 return TC_ACT_OK; 7 }8 __be32* nodeip_ =bpf_map_lookup_elem(&egressip_cache, &iphdr->daddr); 9 if (!nodeip_) { 10 // TOS 0x4 用作未命中标记 11 set_ip_tos(skb, 0, 0x4); 12 return TC_ACT_OK; 13 } 14 struct 出口信息* 出口信息_ =bpf_map_lookup_elem(&egress_cache, nodeip_); 15 if (!出口信息_) { 16 set_ip_tos(skb, 0, 0x4); 17 return TC_ACT_OK; 18 } 19 struct 入口信息结构体* 入站信息_ =bpf_map_lookup_elem(&ingress_cache, &iphdr->saddr); 20 if (!入站信息_ ||!入站信息_complete(入站信息_)) 21 return TC_ACT_OK;

步骤 #2：封装与主机内路由。

```
// 添加外层首部
如果(bpf_skb_调整_空间(skb, 50, BPF_ADJ_ROOM_MAC 地址,
    BPF_F_ADJ_ROOM_FIXED_GSO |
    BPF_F_ADJ_ROOM_封装_L3_IPV4 |
    BPF_F_ADJ_ROOM_ENCAP_L4_UDP |
    BPF_F_ADJ_ROOM_封装_L2(14) |
    BPF_F_ADJ_ROOM_ENCAP_L2_ETH)))
    返回流量控制_ACT_确定;

// 边界检查
...
// 标头包括 50B 的外层报头和 14B 的内层 MAC 报头。
__内置_memcpy(data, 出口信息_->标头, 64);
设置_的长度和 ID（skb, skb->len);
// 设置 UDP 源端口
__u32 hash =bpf_get_hash_recalc(skb);
__be16 源端口 =获取_UDP端口(hash);
bpf_skb_存储_字节(skb, UDP_端口_偏移, &sport,
    大小(sport), 0);
// 数据包重定向
操作 =bpf_重定向(出口信息_->接口索引, 0);
返回操作;
```

B.3.2 入站数据路径

步骤 #1：检查目的地址。

```
1 struct bpf_elf_map SEC("映射") devmap ={ 2 .type =BPF_
MAP_TYPE_HASH, 3 .size_键 =sizeof(int), 4 .size_值 =
sizeof(struct devinfo), 5 .pinning =PIN_GLOBAL_NS, 6 .max_
elem = 8, 7 }; 8 9 struct ethhdr *outer_eth =data; 10 int ifindex =
ctx->接口索引; 11 struct devinfo *devinfo_ =bpf_map_lookup_
elem(&devmap, &ifindex); 12 if (!devinfo_ ||maccmp(outer_eth->
h_dest, devinfo_->mac, ETH_ALEN)) return TC_ACT_OK; 13 //检查
以太网帧是否包含 IP 数据包，并设置 IP 头部指针 14 if(outer_eth->
h_proto != bpf_htons(ETH_P_IP)) return TC_ACT_OK; 15 struct
iphdr *outer_iph =(struct iphdr *)(outer_eth + 1); 16 if(outer_
iph->daddr != devinfo_->ip) return TC_ACT_OK;
```

步骤 #2：缓存检索。

```
1 struct fivetupletuple_; 2 if (parse_5tuple_in(inner_iph,
data_end, &tuple_)) return TC_ACT_OK; 3 struct action *action =
_bpf_map_lookup_elem(&filter_cache, &tuple_); 4 if (!action_ ||!
(action_->入站 & action_->出站)) { 5 set_ip_tos(skb, 50, 0x4); 6
return TC_ACT_OK; 7 } 8 struct ingressinfo* ingressinfo_ =bpf_
map_lookup_elem(&ingress_cache, &inner_iph->daddr); 9 if (!
ingressinfo_ ||!ingressinfo_complete(ingressinfo_)) { 10 set_ip_
tos(skb, 50, 0x4); 11 return TC_ACT_OK; 12 } 13 if (!bpf_map_
lookup_elem(&egressip_cache, &inner_iph->saddr)) return TC_
ACT_OK;
```

步骤 #3：解封装和主机内路由。

```
1 如果(bpf_skb_调整_room(ctx,-50,BPF_调整_空间_MAC 地址, 0))
      返回流量控制_ACT_确定;
2 // 边界检查
3 ...
4 // 设置新的 MAC 地址
5 结构体以太网头部 *eth =数据;
6 __内置函数_memcpy(eth->h_目的MAC地址, 入站信息_->目的MAC地址,
      ETH_ALEN);
7 __内置函数_memcpy(eth->h_源MAC地址, 入站信息_->源MAC地址,
      ETH_ALEN);
8 // 数据包重定向
9 操作 =bpf_重定向_peer(入站信息_->接口索引, 0);
10 返回操作;
```

C 映射大小的计算

根据 §3.1 中描述的缓存定义，缓存条目的大小如下：出口缓存（第一级）为 8 字节，出口缓存（第二级）为 72 字节，入口缓存为 20 字节，过滤器缓存为 20 字节。

为了消除 LRU 机制造成的缓存驱逐，对于每台主机最多有 110 个容器、共 5k 台主机、总计 150k 个容器且每台主机最多有 1M 个并发流的集群，我们需要出口缓存（第一级）有 150k 个条目，出口缓存（第二级）有 5k 个条目，入口缓存有 110 个条目，过滤器缓存有 1M 个条目。

因此，可以按如下方式计算缓存的大小：出口缓存为 8 B * 150k + 72 B * 5k = 1.56 MB，入口缓存为 20 * 110 = 2.2 KB，过滤器缓存为 20 B * 1M = 20 MB。

D 一个示例：理解反向检查的必要性

我们进一步通过一个反例说明反向检查的必要性。

考虑这样一种情形：ONCache 仅在出站数据路径上检查出口缓存。最初，流 f 在连接跟踪中进入已建立状态，并且 f 的所有缓存都已初始化。因此，流 f 的所有数据包都会经过 ONCache 的快速路径。然而，由于流 f 在备用覆盖网络中绕过了连接跟踪模块，其连接跟踪条目最终会在预先设定的时长后过期。之后，如果流 f 因 LRU 机制而从入口缓存中被逐出，它仍可利用出口快速路径，但不再符合使用入口快速路径的条件。理想情况下，我们希望 ONCache 重新初始化流 f的入口缓存。但遗憾的是，流 f 无法在连接跟踪中重新进入已建立状态，因为连接跟踪只有在观察到两个方向上的数据包后，才会将流记录为已建立状态 [2]。因此，流 f 的入口缓存永远无法重新初始化，流 f 也就无法再使用入口快速路径。

E 兼容性讨论

分段卸载。 分段卸载（例如通用分段卸载 (GSO) 和通用接收卸载 (GRO)） [61],使网络接口能够对数据包进行分段和重组。所有 ONCache eBPF 程序的挂载点都是流量控制（TC）。在出站路径上，GSO 在 TC 之后执行 [5]。因此，快速路径不会绕过主机接口的 GSO 及其他硬件分段机制。在入站路径上，GRO 在 TC 之前执行 [5],，不会与 ONCache 产生冲突。因此，ONCache 兼容分段卸载技术。

扩展技术。 Linux 扩展技术，包括接收数据包引导（RPS）和接收流引导（RFS）（以及它们的硬件对应技术——接收侧扩展（RSS）和加速型 RFS（aRFS）） [59],在入口数据路径上采用不同的中央处理器负载均衡策略。它们都与 ONCache 兼容：RSS 和 aRFS 是硬件机制；RPS 和 RFS 则由软件实现，但发生在

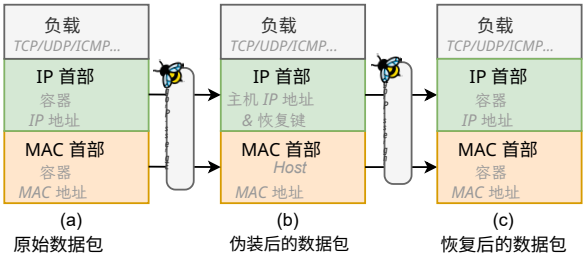

图10：通过基于重写的隧道协议由 *ONCache* 转发的容器数据包的变化。*(a)* 来自容器的原始数据包。*(b)* 伪装数据包。*(c)* 还原后的数据包。

入口路径上的 Ingress-Prog 之前。

校验和卸载。 校验和卸载，包括发送和接收校验和卸载 [10],通常会启用，以减少校验和开销。由于它们是硬件机制，ONCache 与之兼容。

F 基于重写的隧道协议

本节介绍 ONCache 中基于重写的隧道协议如何工作。

基于重写的隧道协议通过重写内层首部，替代传统隧道协议中外层首部的封装。数据路径上对数据包首部的修改如图所示 10。

新的缓存。 出站和入站缓存经过重新设计，以适应基于重写的隧道协议。

出口缓存存储出站路径上使用的数据，结构为 <容器 源目的 *IP* → 主机接口索引、主机 源目的 *IP*、主机 *MAC* 地址、恢复键>[8]。

入口缓存存储入口路径上使用的数据，其结构为 <主机 *sIP* 和恢复键 → *veth*（主机侧）索引、容器 *sdIP*、容器目的*MAC*地址>。为适应后文所述的新初始化过程，入口缓存被划分为两个映射。第一个是入站 IP 缓存： <主机 *sIP* 和恢复键 → 容器 *sdIP*>；第二个是入口缓存： < 容器目的 *IP* → 容器目的*MAC*地址、*veth*（主机侧）索引>。

新的数据包传输路径。 在发送端主机上，Egress-Prog 从出站缓存中检索 <容器 *sdIP* → 主机接口索引、主机 *sdIP*、主机*sdMAC*、恢复键> 。随后，Egress-Prog 通过将容器的源/目的 MAC/IP 地址修改为主机的相应地址，执行伪装 进程。

为确保伪装数据包能在接收端主机上还原，ONCache 还会将恢复键写入数据包中。随后，接收端主机利用恢复键和主机 IP 地址，还原容器源/目的 MAC/IP 地址。恢复键在缓存初始化期间分配，并存储在出口缓存中。用户可以灵活指定数据包中的某个字段来存放恢复键。该字段可以是 IP 首部中的任何可用字段，例如 ID、DSCP 或 Option。恢复键的容量取决于所选字段的位宽。将恢复键写入数据包（即伪装数据包）后，Egress-Prog 会通过 bpf_redirect 或 bpf_redirect_rpeer 将其重定向到主机接口。

在接收端主机上，Ingress-Prog 还原伪装数据包。Ingress-Prog 从入口缓存中检索 <主机 *sIP* 和恢复键 → *veth*（主机侧）索引、容器 *sdIP*、容器 *dMAC*>，并将容器 IP 和 MAC 地址写回标头。还原后，数据包通过 bpf_redirect_peer 重定向到目的 veth。

新的缓存初始化。 MAC 和 IP 地址以及接口索引的初始化仍与原始设计相似（如第 §3.2节所述），并新增了恢复键的分配。恢复键在接收端主机上分配，随后传递给发送端主机。这确保伪装操作仅在接收端主机识别出恢复键后才使用该键。过滤器缓存的初始化与第 §3.2 节所述相同，因此本节不再赘述。ONCache 利用隧道数据包的往返来初始化缓存。该过程如图11所示，包含四个步骤：

步骤 #1（在图 11 1）中标记：遇到缓存未命中时，Egress-Prog 会将数据包传递到备用覆 <容器 *sdIP* → 主机接口索引、主机 *sdIP*、主机 *sdMAC*> 盖网络。随后，数据包在备用覆盖网络中经过处理，到达 Egress-Init-Prog。Egress-Init-Prog 解析标头，并将 <主机 *sIP* 和恢复键 → 容器 *sdIP*> 存储到出口缓存中。它还会为反向流分配一个恢复键，该键以随机或顺序方式选取，并将 <主机 *sIP* 和恢复键 → 容器 *sdIP*> 写入入站 IP 缓存。作为哈希映射，入站 IP 缓存会自然确保恢复键的唯一性。为了将恢复键传递给对等主机，Egress-Init-Prog 会将恢复键写入内层标头中的用户定义字段。最后，数据包被传输到底层网络。

步骤 #2（图 11 2 中标记）：接收端主机收到隧道数据包后，Ingress-Init-Prog 继续进行初始化。它解析数据包中包含的恢复键，并将 <容器 *sdIP* → 恢复键> 存储到出口缓存中。接下来，它将 <容器目的 *IP* → 容器目的*MAC*地址>存储到入口缓存中（<容器目的 *IP* → 虚拟以太网设备（主机侧）索引> 由用户空间守护进程维护）。最后，数据包被转发至目标应用程序。

上述两个步骤填充了两台主机上的出口缓存和入口缓存中一半的字段。随后，该隧道数据包的回复会经过一个类似但方向相反的

[8]sIP = 源 IP 地址；dIP = 目的 IP 地址。同样地，sMAC、dMAC、sdIP 和 sdMAC 也遵循此规则。

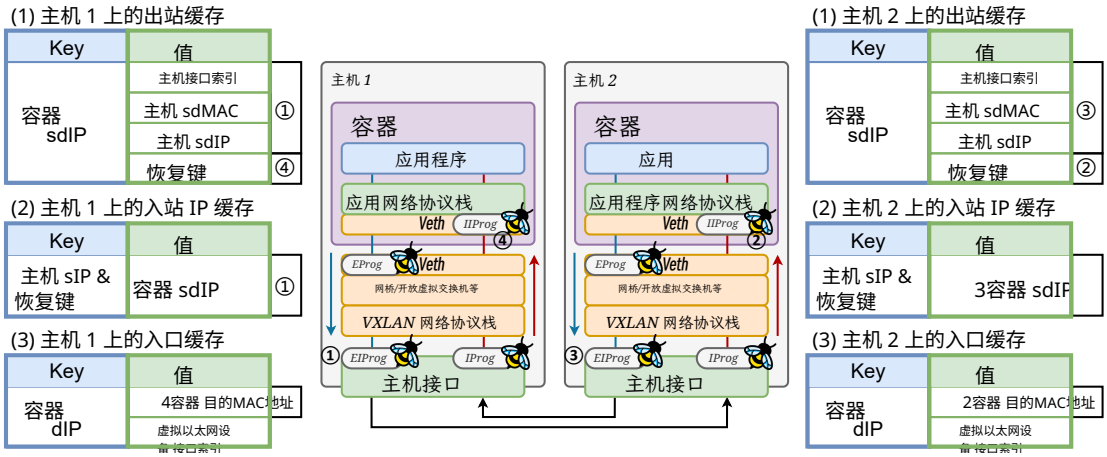

图 11： 采用基于重写的隧道协议的 *ONCache* 缓存初始化过程。1 -4 标记了填充部分缓存的四个步骤。接口索引（*ifidx*）是 *interface index* 的缩写。

结果		ONCache-t	ONCache-r	ONCache-t-r	Host	ONCache
Memcached	延迟	-2.94%	-7.01%	-6.04%	-5.38%	0%
	TPS	+2.97%	+1.74%	+4.32%	+7.42%	0%
	CPU	+1.92%	+2.14%	-1.10%	-6.21%	0%
PostgreSQL	延迟	+0.04%	-1.28%	-5.77%	-2.25%	0%
	TPS	-0.27%	+1.23%	+6.21%	+2.61%	0%
	CPU	-1.86%	-4.61%	-10.36%	+13.17%	0%
HTTP/1.1	延迟	-2.82%	-8.46%	-10.00%	-13.08%	0%
	TPS	+2.78%	+9.09%	+10.90%	+15.10%	0%
	CPU	-5.48%	-11.93%	-14.02%	-11.26%	0%
HTTP/3	延迟	0.00%	+0.16%	+0.16%	+0.04%	0%
	TPS	-0.03%	-0.03%	-0.01%	-0.03%	0%
	CPU	-26.30%	+15.82%	-10.51%	-10.50%	0%

表4： 与 *ONCache* 相比，*ONCache-t*、*ONCache-r*、*ONCache-t-r* 和主机网络的应用程序性能及服务器 *CPU* 利用率（按每秒事务数归一化）。

进程（在图中标记为 1 1 3 4），从而完成整个初始化过程。

G 可选改进的应用评估

本节展示可选改进的应用评估结果。ONCache-r、ONCache-t、ONCache-t-r 和主机网络的评估结果列于表4中。所列值均相对于 ONCache。

应用评估揭示了以下关键发现：(1) 除 HTTP/3 外，基于重写的隧道协议（ONCache-t）和 bpf_redirect_rpeer（ONCache-r）均提升了所有应用的延迟和每秒事务数性能。它们还降低了每笔事务的 CPU 利用率，适用于 PostgreSQL 和 HTTP/1.1。(2) 启用这两项可选改进后，ONCache-t-r 通常在所有应用中实现最佳的延迟、每秒事务数和 CPU 利用率，HTTP/3 除外。值得注意的是，ONCache-t-r 的性能与主机网络十分接近。

由于 Nginx 对 QUIC 的实验性支持存在限制，不同实现中 HTTP/3 的绝对结果值略有差异。因此，HTTP/3 的评估结果并无定论。