中国科学技术大学
University of Science and Technology of China

The Design and Implementation of Open vSwitch

NSDI 2015

授课教师：赵功名

中国科大计算机学院

2025年秋·高级计算机网络

Outline

I. Introduction

II. Design Constraints and Rationale

III. Design

IV. Flow Cache Design

V. Caching-aware Packet Classification

VI. Cache Invalidation

VII. Evaluation

Rise of Virtualization

虚拟化已经重塑了计算的方式，许多数据中心实现了全面的虚拟化来提供：

- 快速的资源配置
- 向云端的拓展
- 故障恢复期更高的可用性

数据中心网络中，出现了新型的网络**接入(access)层**，其中大多数网络接口是**虚拟**的，而不是物理接口。

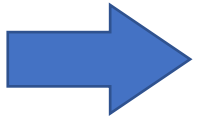
这些虚拟机(VM)的第一跳交换机从物理机房迁移到了**虚拟机管理程序(hypervisor)**中。

需要软件实现的虚拟交换机(vSwitch)来为虚拟服务器提供网络功能。

Traditional vSwitch

传统的虚拟交换机，例如早已在Linux内核中实现的Linux Bridge：

- 只关心最基本的**网络连通性**，即让VM能连上原来的物理L2网络
- 只是模仿机架交换机(ToR)的功能，把物理交换机的L2功能用软件实现
- 只负责把VM的流量转发到物理交换机，更多的网络功能仍由**物理交换机/路由器**实现



随着虚拟化负载的激增，这样的问题出现了许多**问题**：

- 由于vSwitch只实现桥接的功能，VM本质上仍在连接在物理交换机上，新的工作负载需要重新配置和准备**物理网络**，**减慢了工作负载的部署速度**
- 工作负载与物理L2网段的**紧耦合**，严重限制**工作负载的迁移能力(mobility)**和**底层网络的可扩展性(scalability)**

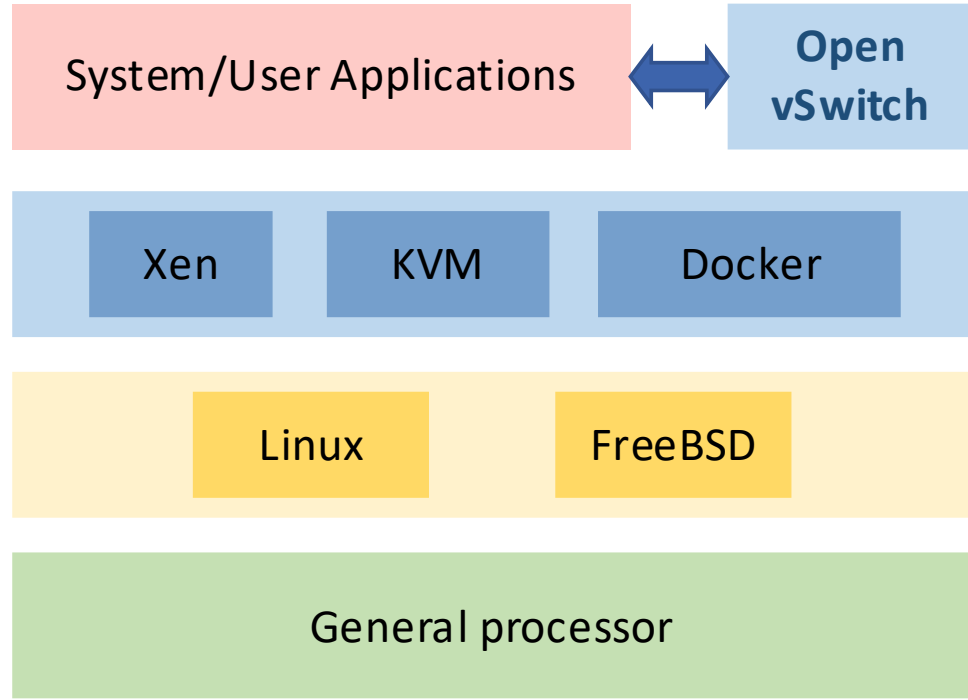
Network Virtualization

因此，计算的虚拟化需要**网络的虚拟化(network virtualization)**。

- VM**网络功能**(路由、ACL、安全策略、隔离等)的提供者从物理交换机变为**虚拟交换机**
- **物理网络**只在hypervisor间转发IP隧道封装的流量

从而，

- 让虚拟网络从底层的物理网络中**解耦**
- 利用通用处理器的灵活性，虚拟交换机可以为VM，租户和网络管理员提供与专用物理网络完全**一致的逻辑网络抽象、服务和工具**。



Capable Virtual Switch

网络虚拟化需要**功能强大的虚拟交换机**。

传统的虚拟交换机	可以实现网络虚拟化的虚拟交换机
➢ 使用的是静态的、主要是硬编码的转发处理流程 ➢ 对于提供基本L2连接足矣	➢ 转发功能必须能够基于**每一个虚拟端口**进行定制* ➢ 多个虚拟机管理程序（hypervisor）之间实现这些抽象时，还非常受益于**细粒度的集中式协调**

*由于虚拟交换机需要承担更多的网络功能，例如多租户隔离、虚拟网络拓扑、策略控制(ACL，QoS)、VXLAN等overlay功能，每个虚拟端口(vPort)可能属于不同的租户，采用不同的网络策略，因此vSwitch必须支持**逐端口**，**细粒度**的功能配置，而不是像传统网桥只提供基础的L2转发。

Key Elements

本文讲述Open vSwitch的设计和实现，主要围绕两个方面/挑战：

- **高性能**：部署Open vSwitch的生产环境往往需要高性能
- **通用性**：网络虚拟化对可编程性有要求

一般的网络设备（无论软件/硬件）通常通过专用化获得高性能，而OVS则追求灵活性和通用性。本文关注如何在不牺牲通用性（适应不同平台/hypervisor/工作负载）的前提下实现高性能。

Outline

I. Introduction

II. Design Constraints and Rationale

III. Design

IV. Flow Cache Design

V. Caching-aware Packet Classification

VI. Cache Invalidation

VII. Evaluation

Design Constraints and Rationale

虚拟交换机和传统网络设备工作的环境存在巨大差异。

下面介绍这些差异导致的**限制**和**挑战**。

并由此说明OVS设计背后遵循的**基本原则**和独特的亮点。

Resource Sharing

OVS和VM共享CPU资源

传统网络设备

传统网络设备往往采用专用硬件资源保证在**最坏**情况(worst case)下也能保持**线速**(line rate)。

虚拟交换机

- 虚拟交换机和VM**共享**宿主机的CPU资源
- **节约资源**(resource conservation)更重要：保证最坏情况下的线速不如最大化hypervisor用于运行用户工作负载的可用资源。即，应优化**通常情况**(common case)，而非最坏情况
- **设计**：因此，往往大量采用**流缓存**(flow caching)或其他形式的缓存，来在通常情况(命中率高)下降低资源占用
- **异常情况**的处理：端口扫描、P2P会合服务器、网络监控等会产生异常流量，这些异常情况也需要支持，但无需保证线速

Placement

OVS位于网络边缘，即带来了简化，也带来了复杂性

简化

OVS位于拓扑的叶节点，且和hypervisor，VM形成一个整体。这种位置消除了许多网络问题。

复杂性

- 让扩展(scaling)变得复杂
- 在hypervisor间P2P隧道形成的网状结构(mesh)中，一个虚拟交换机有几千个对端(peer)虚拟交换机也是常事
- 当远端有VM启动、关机或迁移时，虚拟交换机会收到转发状态更新；在大规模部署中，转发状态可能会持续高频变更
- **设计**：Open vSwitch分类(classification)算法，实现O(1)更新

OVS还有三个独特的需求

- **OpenFlow交换机**
- 应对更高分类负载
- 开源，多平台

OpenFlow交换机

- OVS从设计之初就是**OpenFlow交换机**
- 传统的虚拟交换机所采用**功能固定的数据路径**(feature datapath)：采用类似于硬件ASIC的固定包处理流水线，只能在预定义的范围内进行配置
- OVS不与某个专用的、纵向紧耦合(tightly vertically integrated)的网络控制栈绑定，而是通过OpenFlow实现可重新编程能力

OVS还有三个独特的需求

- OpenFlow交换机
- **应对更高分类负载**
- 开源，多平台

应对更高分类负载

- OpenFlow的灵活性在SDN中是不可或缺的，但是在高级用例（如网络虚拟化）中，会带来**更长的包处理流水线**，从而带来比传统虚拟交换机**高得多的分类负载**
- **设计**：为了避免OVS相比于其他虚拟交换机消耗更多的hypervisor资源，必须实现**流缓存**(flow cache)绕开包处理流水线

OVS还有三个独特的需求

- OpenFlow交换机
- 应对更高分类负载
- **开源，多平台**

开源，多平台

- **闭源**虚拟交换机都运行在**单一环境**中
- Open vSwitch 是**开源和多平台**的，所处的环境往往由用户自行选择（**操作系统**发行版与 **hypervisor** 的组合）
- **设计**：必须高度模块化和可移植的设计

Outline

I. Introduction
II. Design Constraints and Rationale
III. **Design**
IV. Flow Cache Design
V. Caching-aware Packet Classification
VI. Cache Invalidation
VII. Evaluation

Two Major Components

OVS有两个主要组件负责指挥数据包转发。

ovs-vswitchd	kernel datapath
运行在**用户态**的守护进程	**内核**模块
在不同的操作系统和运行环境中基本**保持一致**	通常需要针对宿主操作系统进行**专门实现**，以获得更高的性能
负责**控制逻辑**、管理规则、执行OpenFlow、构建pipeline	负责真正的**数据路径**，在内核中高速转发数据包

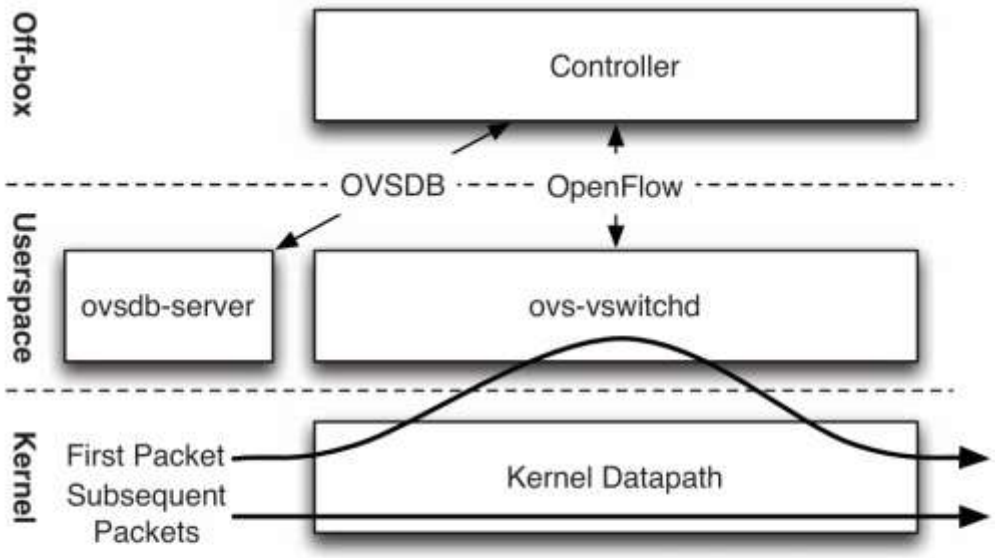


Figure 1: The components and interfaces of Open vSwitch. The first packet of a flow results in a miss, and the kernel module directs the packet to the userspace component, which caches the forwarding decision for subsequent packets into the kernel.

Packet Forwarding

ovs-vswitchd和datapath共同指挥包转发。

当datapath从物理NIC或虚拟VM的虚拟NIC收到一个数据包：

- 如果ovs-vswitchd**已经指示**datapath如何处理，那么datapath模块就会直接遵循这些指令（actions）
- 如果datapath**尚未收到**相关指令，它就会将数据包上送给ovs-vswitchd。在用户态中，ovs-vswitchd 会决定数据包应如何处理，然后将带有**处理策略**的数据包返回给 datapath。

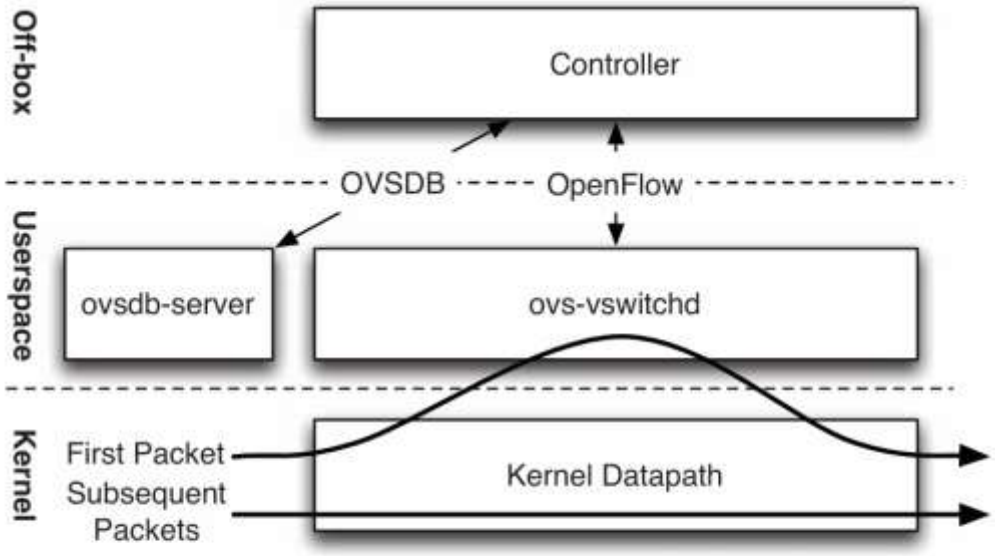


Figure 1: The components and interfaces of Open vSwitch. The first packet of a flow results in a miss, and the kernel module directs the packet to the userspace component, which caches the forwarding decision for subsequent packets into the kernel.

Flow Caching

OVS的流缓存(flow caching)

OVS的datapath中有**两级**缓存：

- **微流缓存 (microflow caching)**：按**每个传输连接**缓存对应的转发决策；**精确匹配**
- **大流缓存 (megaflow caching)**：为超越单个连接的**更大流量集合(traffic aggregates)**缓存转发决策；**wildcard 匹配**

OpenFlow

OVS通常被用作**SDN交换机**，通过**OpenFlow**控制转发。

在OVS中，ovs-vswitchd：

- 从SDN控制器**接收OpenFlow流表**
- 将来自内核datapath的数据包与这些 OpenFlow 流表进行**匹配**，**收集**需要应用的**动作**（actions）
- 将结果**缓存**到内核datapath中

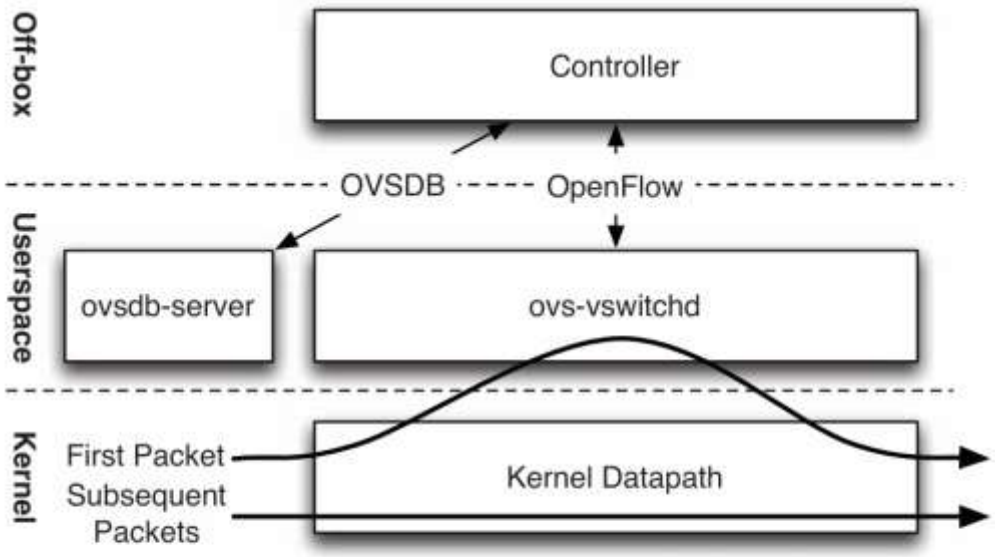


Figure 1: The components and interfaces of Open vSwitch. The first packet of a flow results in a miss, and the kernel module directs the packet to the userspace component, which caches the forwarding decision for subsequent packets into the kernel.

OpenFlow

这样的方式实现了**SDN控制器**和**datapath**的解耦：

- **内核datapath模块视角**：无需了解OpenFlow的具体协议细节，从而使datapath设计更加简化
- **控制器视角**：缓存机制以及用户态与内核态的分离都**不可见**：每个数据包都会访问一系列OpenFlow流表，找到符合该数据包条件的最高优先级流项，并执行其对应的OpenFlow动作

OVS的**流编程模型**在很大程度上决定了它所能支持的**应用场景**。为此，OVS针对**网络虚拟化需求**，对标准OpenFlow做了大量**扩展**。

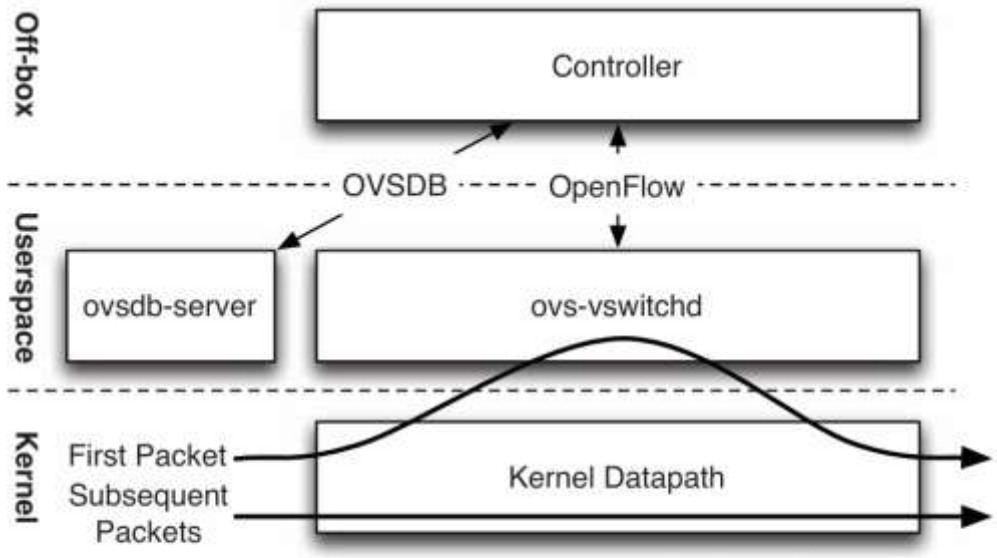


Figure 1: The components and interfaces of Open vSwitch. The first packet of a flow results in a miss, and the kernel module directs the packet to the userspace component, which caches the forwarding decision for subsequent packets into the kernel.

Packet Classification

问题

OpenFlow交换机中，**报文分类(packet classification)**指的是对进入交换机的每个数据包，判断该数据包匹配哪条**流**(flow)，应该执行什么**动作**(action)。

使用通用处理器进行算法式的数据包分类**开销**非常大，尤其是在OpenFlow中，因为OpenFlow匹配的**泛用性(generality)**：需要测试众多字段（Eth地址，IPv4/6地址，ingress/egress端口等等）的**任意组合**。

Packet Classification

OVS采用**元组空间搜索(tuple space search)**分类器。

做法：对于每一种**匹配形式(form of match)**，即**匹配字段的组合**维护一个**哈希表。**

例子

OpenFlow记录了两条流：

- Flow A: 根据(src_eth, dst_eth)两个字段匹配pkt
- Flow B: 根据(src_eth, dst_eth)两个字段匹配pkt

由于两条流的匹配字段一致，即只有一种形式的匹配(form of match)，交换机只需要维护一个哈希表，即(src_eth, dst_eth)，并对进入的每一个数据包的这两个字段尝试匹配。

Flow	Form of Match
A	(src_eth, dst_eth)
B	(src_eth, dst_eth)

Hash Table	Key	Corresponding Flows
1	(src_eth, dst_eth)	A, B

Packet Classification

OVS采用**元组空间搜索(tuple space search)**分类器。

做法：对于每一种**匹配形式(form of match)**，即**匹配字段的组合**维护一个**哈希表。**

例子

现在新增了一条流C，匹配的字段是(src_eth, src_ip)，由于这是一种新的匹配形式，需要维护一个新的哈希表。

Flow	Form of Match
A	(src_eth, dst_eth)
B	(src_eth, dst_eth)
C	(src_eth, src_ip)

Hash Table	Key	Corresponding Flows
1	(src_eth, dst_eth)	A, B
2	(src_eth, src_ip)	C

Packet Classification

OVS采用**元组空间搜索(tuple space search)**分类器。

做法：对于每一种**匹配形式(form of match)**，即**匹配字段的组合**维护一个**哈希表。**

例子

当一个流到达的时候，需要对两个哈希表进行查找：

- 如果都不匹配，流表中不包含匹配
- 如果其中一个匹配，则找到对应的流
- 如果两个都找到了匹配，则选取优先级较高的流

随着流数量的增加，匹配形式也随之增加；分类器进行扩展，为每种匹配形式维护一个哈希表，并且分类器的一次查找需要查找所有哈希表。

Flow	Form of Match
A	(src_eth, dst_eth)
B	(src_eth, dst_eth)
C	(src_eth, src_ip)

Hash Table	Key	Corresponding Flows
1	(src_eth, dst_eth)	A, B
2	(src_eth, src_ip)	C

Packet Classification

元组空间搜索的**查找(lookup)**复杂度相比于SOTA方案较差，但是对于实际中的大多数流表表现很好，并且相对于决策树(decision tree)分类算法有三个独特优势：

- 支持常数时间的**高效更新(update)**，一次更新对应一次哈希表操作，并且对于虚拟化场景而言，中心化控制器需要高频更新（增删流）以反映整个数据中心的变化。
- 可以**泛化**到任意数量的头部字段，无需算法上的修改。
- 元组空间搜索的**空间开销**随流数量**线性**变化。

Packet Classification

除了上面提到的查找复杂度大以外，包分类器的开销被复杂的SDN控制器使用的**大量流表**而进一步放大。例如，VMware的网络虚拟化控制器所安装的流表，其数据包处理流水线中每个数据包至少需要进行约 15 次流表查询。

使用长流水线主要有两个原因：

- 如果通过表的交叉组合(cross-producting)来减少阶段数量，会显著增加流表规模*
- 开发者更倾向于将流水线设计模块化。

所以，**减少单个数据包的哈希查找次数**尤为关键。

*例如，对以下两个阶段就行交叉组合，表项从n1+n2变为n1*n2.

Phase 1 Fields	Flow
(f1, f2)_1	flow_1
…	
(f1, f2)_n1	flow_n1

×

Phase 2 Fields	Flow
(f3, f4)_1	flow_(n1+1)
…	
(f3, f4)_n2	flow_(n1+n2)

→

Phase 1*2 Fields	Flow
(f1, f2, f3, f4)_1	flow_1
…	
(f1, f2, f3, f4)_(n1*n2)	flow_(n1+n2)

OpenFlow as a Programming Model

多次提交 (resubmit)

为解决 cross-producting 的问题，Open vSwitch 引入了一个扩展动作 **resubmit**，允许数据包在转发表之间**多次查表**（甚至重复查同一张表），并将所有匹配到的动作**合并**。

这样就可以将字段 A 的 n_1 条规则放在一张表、字段 B 的 n_2 条规则放在另一张表，从而避免 $n_1 \times n_2$ 的规则爆炸。

同时，resubmit 也支持基于某些字段值进行**多分支**的编程方式。

Phase 1 Fields	Flow
(f1, f2)_1	flow_1
…	
(f1, f2)_n1	flow_n1

Phase 2 Fields	Flow
(f3, f4)_1	flow_(n1+1)
…	
(f3, f4)_n2	flow_(n1+n2)

Outline

I. Introduction
II. Design Constraints and Rationale
III. Design
IV. Flow Cache Design
V. Caching-aware Packet Classification
VI. Cache Invalidation
VII. Evaluation

Microflow Caching

all-in-kernel 的困境

OVS开始开发时，所有OpenFlow处理都放在了**一个内核模块**中：完成数据包的接受，分类，修改，转发。但后来这种方法不再适用：

- 在内核中开发、分发和更新内核模块都相对困难
- 在内核中实现OpenFlow不可能被上游 Linux 接受

因此需要把大部分复杂功能移入用户空间

Microflow Caching

解决方案：微流缓存(microflow caching)

微流缓存：每个**缓存项**都对 OpenFlow 支持的**所有数据包头字段**进行**精确匹配**

内核模块可以用一个简单的**哈希表**来实现，而不再需要一个支持任意字段和掩码的复杂通用数据包分类器。

问题：

- 缓存**粒度细**，最多只能匹配到**单个传输连接**的流，miss率高
- miss时将数据包转发到用户态，**内核态-用户态切换代价高**

性能的关键维度：**流建立时间（flow setup time）**：即内核向用户态报告一次微流未命中并等待用户态回复所需要的时间。

Microflow Caching

微流缓存的局限

OVS采取了多种**优化手段**：

- 通过对同时到达的流建立请求进行**批处理**，减少系统调用次数
- 将流建立的负载分散到多个**用户态线程**上，以利用多核 CPU 的优势
- 实现multiple-reader, single-writer的无阻塞流表结构

然而，简单的缓存设计无论如何都难以满足：

- **延迟敏感 (latency-sensitive)** 的应用
- 大量**短生命周期**连接

因此需要重新设计内核中的**缓存结构**。

Megaflow Caching

Megaflow caching: 支持**通用匹配**，因此支持为比单个连接**更大粒度**的流量缓存转发决策。

Megaflow caching采用**元组空间搜索（TSS）**，为每个匹配模式维护一个哈希表。

Megaflow caching类似通用的Openflow表，但更加轻量化：

- **没有优先级**：找到一个匹配项即可，不必查找所有哈希表
- 只需要一个megaflow分类器，**无流水线**

Megaflow Caching

megaflow caching 的局限

如果有n个哈希表，假设每个哈希表命中的概率相同，则每个packet进行哈希查找次数：

- 命中：期望值为 (n+1)/2
- 未命中：n

对于一个流中的每个packet都进行如此多次数的查找，代价太大！

解决方案：保留 **microflow 缓存**作为**一级缓存** (first-level cache)。

该缓存是一个哈希表，将 microflow 映射到其对应的 megaflow。

一个微流的第一个数据包会通过内核 megaflow 表，需要搜索分类器；但随后该微流中的数据包即可通过精确匹配缓存快速定位到正确的 megaflow 项。

效果：megaflow 的成本从“每包一次”降低到“每微流一次”。

Megaflow Caching

为了减少开销，OVS 的用户态守护进程以**增量式、反应式（incremental and reactive)** 方式计算 megaflow 缓存项

- 用户态流表处理数据包时会记录分类算法过程中使用到的**字段位** (packet field bits); 生成的 megaflow 条目匹配所有在决策中被使用到的字段
- megaflow 缓存的填充**由到达的数据包驱动**，并随着流量聚合的变化而动态调整：新的条目被填充，而旧的条目被移除。

Outline

I. Introduction
II. Design Constraints and Rationale
III. Design
IV. Flow Cache Design
V. **Caching-aware Packet Classification**
VI. Cache Invalidation
VII. Evaluation

Problem

OVS的用户态程序在其 OpenFlow 表中处理一个数据包时，会追踪在转发决策过程中使用到的**数据包字段位**，从而构造 megaflow 条目。

- 对于**简单**的OpenFlow表十分有效
- 额外字段的引入可能导致megaflow缓存性能下降

Case 1:

- OpenFlow只检查L2字段
- 现有**端口扫描**程序，L2字段不变，L4端口字段频繁变化

结果：生成的 megaflow 会将L3，L4的字段mask掉，得到**近乎理想**的缓存命中率。

Case 2:

- 在 Case 1 的基础上，有一个流匹配 **TCP的端口号**

结果：每个 megaflow 必须检查数据包的 TCP 端口，导致端口扫描程序的megaflow变为microflow的粒度，**性能严重下降。**

Problem

寻找最优的，粒度最大(least specific)的 megaflow 的在线算法很难找到。

OVS将重点放在生成**更好的近似方案**上。

字段过少可能导致错误的包转发，因此近似策略倾向于匹配比必要更多的字段（以确保正确性）：

- 5.2：按**优先级**查找，减少**查找次数**和涉及的**字段数量**
- 5.3：分**阶段**查找，提前排除不匹配tuple
- 5.4：利用**前缀**在 IP 匹配中生成更粗粒度的 megaflow

Tuple Priority Sorting

优化1：优先级排序

问题：在元组空间分类器中进行查找的时候需要搜索**所有**元组，即使已经在前面的元组中找到了匹配，也需要继续查找，以防有**优先级更高的流**也可以匹配。

方案：对于每个元组T，追踪其中流的**最高**优先级T.pri_max。

在查找的时候，按照最高优先级**降序**逐元组查找。

如果发现已经匹配的流F的优先级F.pri高于元组的最高优先级，则可以返回结果。因为后面的元组不可能存在更高优先级的流了。

```
function PRIORITYSORTEDTUPLESEARCH(H)
    B ← NULL        /* Best flow match so far. */
    for tuple T in descending order of T.pri_max do
        if B ≠ NULL and B.pri ≥ T.pri_max then
            return B
        if T contains a flow F matching H then
            if B = NULL or F.pri > B.pri then
                B ← F
    return B
```

Figure 2: Tuple space search for target packet headers H, with priority sorting.

Tuple Priority Sorting

在实际场景中的合理性

同一元组包含的流往往用于相似的目的，优先级也相同。

例如，在VMware的NVP控制器中，包含29个元组，其中26个元组中的流的优先级完全相同。

在这26个元组中，如果找到了匹配的流，则算法可以立即停止。

Staged Lookup

优化2：分阶段查找

问题

元组空间查找对每个元组进行哈希查找，megaflow必须匹配元组中的**每一位**。

当元组中包含**随流频繁变化**的字段（如TCP端口）时，生成的 megaflow 粒度几乎和 microflow 一样，只能匹配某个特定的 TCP 流。

优化方向

如果只需要对元组的**一部分字段**进行匹配，就可以提前判断这个元组**不可能匹配**，则生成的 megaflow 只需要匹配这几个字段，而不是tuple中的全部字段。

Staged Lookup

分阶段查找

将字段按流量的**粒度从粗到**细静态划分为四组：

metadata（例如交换机入口端口）、L2、L3 和 L4。

我们把每个 tuple 从单个哈希表变为包含**四个哈希表**的数组，称为 **stages**：

- 只包含 metadata 字段的哈希表
- 包含 metadata + L2 字段的哈希表
- 包含 metadata + L2 + L3 字段的哈希表
- 包含所有字段（metadata + L2 + L3 + L4）的哈希表（相当于之前的单个哈希表）

对一个 tuple 的查找会**依次**搜索这些 stage。如果某个 stage 查找失败，则整个 tuple 查找失败，而且 megaflow 只需要匹配到当前 stage 所包含的字段。

Staged Lookup

划分思想：

越“**内层**”的头部，其**变化通常越频繁**。例如，在 L4 层，TCP 的源端口和目的端口会随着每个连接而变化；而在 metadata 层，入口端口的数量通常较小且基本固定。

因此四个哈希表 megaflow 的粒度越来越细。

开销分析：

看似每个元组的查找从一次查找变成了多次，会导致分类速度变慢，但测量结果显示：实际上分类速度在实践中反而略有提升，因为当查找在较早的 stage 就失败时，分类器就不需要计算 tuple 中所有字段的**完整哈希值**。而 profiling 显示：哈希计算本身是一个显著的开销。

Staged Lookup

例子

NVP 控制器使用 OVS 来实现多个**相互隔离的逻辑数据路径**。每个逻辑数据路径都是独立配置的。

假设某些逻辑数据路径配置了基于 L4（例如 TCP 或 UDP）端口号的 ACL，用于允许或拒绝某些流量。

然而在这个优化之前，由于分类器的查找必须经过一个会匹配 L4 端口的 tuple，所有生成的 megaflow **都被迫匹配 L4 端口**。

通过这个优化，对于没有 L4 ACL 的逻辑数据路径，其流量在前面三个（或更少）stage 就可以确定“不匹配”该 tuple，因此其 megaflow 不再需要匹配 L4 端口。

优化3：前缀追踪

问题

在 OpenFlow 中，许多流会匹配 IPv4 或 IPv6 的子网，如果不同流使用**不同长度的子网掩码**，那么生成的 megaflow 将会匹配**最长**的那个前缀。

例子

OVS正在为一个目标地址为 10.5.6.7 的数据包构造 megaflow。

OpenFlow流表中有匹配子网 10/8 和主机 10.1.2.3/32的流。

理论上可以安全地安装一个匹配 10.5/16 的 megaflow（因为只需要看前16位就可以确定该数据包属于匹配子网10/8的流，而不属于匹配主机10.1.2.3/32的流）。

但是没有优化的情况下，OVS 会安装一个 10.5.6.7/32 的 megaflow。

Prefix Tracking

通过 trie 结构对 IPv4 和 IPv6 的前缀匹配进行优化

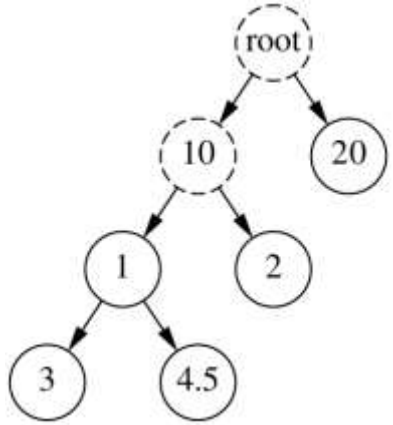
如果某个字段上存在 IP 匹配，分类器在进行 tuple space 搜索之前，会对该字段执行一次**最长前缀匹配（LPM）**查找，用来：

- 确定生成 megaflow 所需要的最大前缀长度
- 确定哪些 tuple 可以被完全跳过，而不影响匹配正确性。

OpenFlow规则，某几个流在某IPv4字段上匹配：

20	/8
10.1	/16
10.2	/16
10.1.3	/24
10.1.4.5	/32

生成的 trie 树：



注：例子中为了方便表示，IP地址采用8-bit组作为单元，但实际实现中以单个bit作为单元

Prefix Tracking

通过 trie 结构对 IPv4 和 IPv6 的前缀匹配进行优化

为了确定需要匹配的位数，OVS根据数据包 IP 地址，从 trie 的根节点向下遍历。

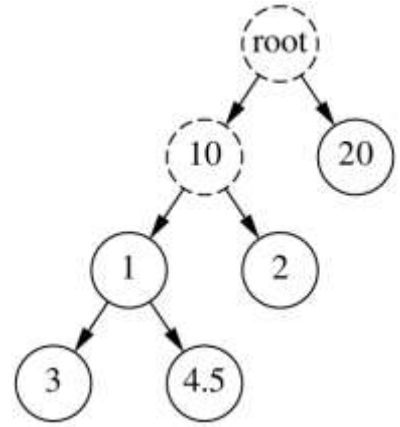
- 如果遍历最终到达一个**叶子节点**，那么 megaflow 就不需要再匹配该地址剩余的位。
- 反之，如果遍历在某一层**无法继续**（即 IP 地址的某些位找不到对应的子节点），那么 megaflow 必须匹配到包含这些无法匹配位为止的全部地址位。

1. 10.1.3.5抵达叶节点3，megaflow安装为10.1.3/24
2. 10.3.5.1在分支节点10无法继续，安装为10.3/16

trie树能让OpenFlow跳过某些tuple的查找：

3. 10.1.6.1在分支节点1处无法继续，则该字段IP地址长度超过16的tuple不可能匹配，无需查找

20 /8
10.1 /16
10.2 /16
10.1.3 /24
10.1.4.5/32



注：例子中为了方便表示，IP地址采用8-bit组作为单元，但实际实现中以单个bit作为单元

Prefix Tracking

通过 trie 结构对 IPv4 和 IPv6 的前缀匹配进行优化

每个节点假设包含以下成员：

- bits：该节点对应的地址比特
- left / right：节点的左右子节点
- n_rules：该节点上挂载的规则数量（如果节点只是为了支撑子节点存在，则为 0）

算法返回：

- 必须匹配的位数（用于改进 megaflow 的掩码）
- 一个位数组（bit-array）：其中 0 位表示对应 prefix 长度的 tuple 可以跳过搜索

```
function TrieSearch(value, root)
    node ← root, prev ← NULL
    plens ← bit-array of len(value) 0-bits
    i ← 0
    while node ≠ NULL do
        c ← 0
        while c < len(node.bits) do
            if value[i] ≠ node.bits[c] then
                return (i + 1, plens)
            c ← c + 1, i ← i + 1
        if node.n_rules > 0 then
            plens[i − 1] ← 1
        if i ≥ len(value) then
            return (i, plens)
        prev ← node
        if value[i] = 0 then
            node ← node.left
        else
            node ← node.right
    if prev ≠ NULL and prev has at least one child then
        i ← i + 1
    return (i, plens)
```

Figure 3: Prefix tracking pseudocode. The function searches for *value* (*e.g.*, an IP address) in the trie rooted at node *root*. It returns the number of bits at the beginning of *value* that must be examined to render its matching node unique, and a bit-array of possible matching lengths. In the pseudocode, $x[i]$ is bit i in x and $\text{len}(x)$ the number of bits in x.

Prefix Tracking

通过 trie 结构对 IPv4 和 IPv6 的前缀匹配进行优化

本算法的初衷是优化优化最长前缀匹配（LPM）查找，但即使是在没有流显式匹配某个IP前缀时，也可以优化megaflow的质量。

OpenFlow为了实现**LPM**，具有**更长前缀**的流必须拥有**更高的优先级**。

但是这样会导致致生成的 megaflow 必须按照流表中**最长前缀**来**取消通配**（unwildcard）IP 地址位。

而Prefix tracking算法可以**提前确定**数据包不属于某规则，从而防止某些**只应用于特定主机的策略**（例如高优先级 ACL）迫使所有 megaflow 都必须匹配完整的 IP 地址。

Classifier Partitioning

优化4：分类器划分

通过跳过那些不可能匹配的 tuple，还可以进一步减少 tuple space 搜索的次数。
OpenFlow 支持在数据包通过 classifier 的过程中设置和匹配 **metadata 字段**。

- 基于某个特定的 metadata 字段对 classifier 进行**分区**
- 如果数据包该字段上与某个 tuple 中的值完全不匹配，那么这个 tuple 会被直接跳过。

NVP（VMware Network Virtualization Platform）通常将 classifier 中的每一次查找配置成类似流水线的一“阶段”。**这些阶段会匹配一组固定的字段**，与 tuple 的概念相似。通过将当前流水线阶段编号存入一个专用的 metadata 字段，NVP 为 classifier 提供了一个提示，使其能够高效地只查找与当前阶段相关的 tuples。

例如，如果一个流只检查L2相关字段，
则可以通过检查metadata直接跳过L3
和ACL相关的tuple。

理解这部分内容需要联系5.3分阶段查询和3.3和resubmit相关的内容。

Outline

I. Introduction
II. Design Constraints and Rationale
III. Design
IV. Flow Cache Design
V. Caching-aware Packet Classification
VI. Cache Invalidation
VII. Evaluation

Difficulty of Cache Management

缓存带来的另一面问题是：**如何管理缓存的复杂性。**

在 Open vSwitch 中，缓存可能因为多种原因需要**更新。**

理想状态：OVS可以精确识别出在某些事件发生后需要更新的那些 megaflow

- **对于某些类型的事件，这一点很容易做到。**例如，某MAC地址的端口发生变化时，更新使用了该MAC地址的 datapath flow 即可，
- **但由于 OpenFlow 模型的高度通用性，在其他情况下要做到精确识别就很困难。**例如，向某个 OpenFlow 表添加一条新的流时：任何匹配了该表中**优先级低于新流**的 megaflow，都有**可能**因此需要改变其行为。

结论：在一般情况下，要做到精确识别并不现实。

Two Group Method

早期方案：将可能影响数据通路的变化分为两类：

1. 影响范围过大、无法精确识别需要更新哪些 datapath flows
2. 影响范围可以被缩小到特定 datapath flows，如MAC 学习表的变化

Case 1：必须检查**所有** datapath flows 是否需要更新

每个 flow 都必须像最初构建它时那样，再次通过 OpenFlow 流表进行一次完整处理，然后将生成的新动作与目前安装在 datapath 中的动作进行比较。

问题：当时的 Open vSwitch 是 **单线程** 的，重新检查所有 datapath flows 所花费的时间会**阻塞**新数据包对应的新 flow 的创建流程（flow setup）。这会使这些 packets 的 flow setup **延迟变得很高。**

应对方案：限制 datapath 中缓存的 flow 数量：~2500条。

Two Group Method

早期方案：将可能影响数据通路的变化分为两类：

1. 影响范围过大、无法精确识别需要更新哪些 datapath flows
2. 影响范围可以被缩小到特定 datapath flows，如MAC 学习表的变化

Case 2：使用一种称为 **tags（标签）** 的技术进行优化

- 对于每一种可能需要触发 megaflow 更新的属性，OVS 都会分配一个 tag。
- 如果属性发生变化，OVS 会把对应 tag 加到一个“变化 tag 集合”里。OVS 会分批扫描整个 megaflow 表，根据tag检查 megaflow 是否需要更新

问题：OVS使用 **Bloom 过滤器**实现 tag，随着控制器变得复杂，tag数量的增加导致**假阳性** (false positive，某 megaflow被误认为包含某个变化的tag)，导致tag方案的效率极低。再OVS 2.0版本中，tag被取消，状态变化时会检查整张 datapath flow表。

Two Group Method

将可能影响数据通路的变化分为两类：

1. 影响范围过大、无法精确识别需要更新哪些 datapath flows
2. 影响范围可以被缩小到特定 datapath flows，如MAC 学习表的变化

Case 2：OVS 2.0方案：**用户态多线程**

- Flow setup多线程：流建立不需要等待重新验证，降低延迟
- 缓存驱逐多线程：缓存驱逐速度和流的建立速度匹配，从而防止datapath flow迅速填满

效果：极大增大了内核缓存的最大大小：~200,000条

OVS驱逐一段时间不使用的datapath flow，通过控制驱逐的阈值使内核缓存尽可能保持在最大大小以内。

Outline

I. Introduction
II. Design Constraints and Rationale
III. Design
IV. Flow Cache Design
V. Caching-aware Packet Classification
VI. Cache Invalidation
VII. Evaluation

Performance in Production

作者在**生产环境 (production)** 和**微基准测试 (microbenchmark)** 两种环境下对OVS进行了测试。

生产环境

- 一座大型商用多租户数据中心中，各个hypervisors在 **24 小时**内的 Open vSwitch 性能数据
- 包含了来自 **1000 多台**运行OVS的hypervisor的统计信息，这些统计数据每 **10 分钟**轮询一次

缓存大小

数据：观测期间最小、平均与最大流条目数量的 CDF

观察：有 50% 的 hypervisor 的平均 flow 数量不超过 107 条。最大 flow 数量的 99 百分位也只有 7,033 条。

结论：实际环境中 小规模的 megaflow 缓存已经足够

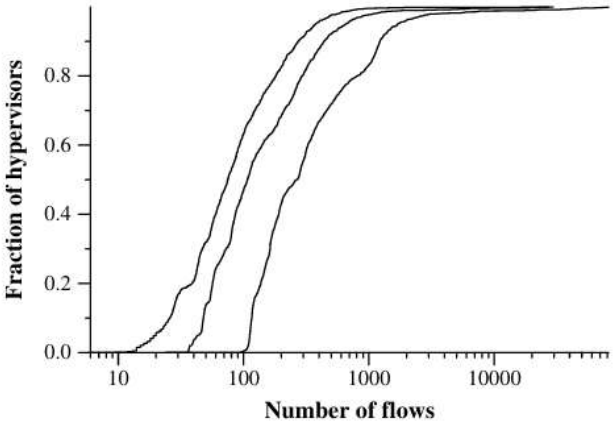


Figure 4: Min/mean/max megaflow flow counts observed.

Performance in Production

缓存命中率

数据：缓存命中率（25%流量最少、整个、25%流量最多的测试区间）；缓存命中和miss的速率

观察：总体命中率达到了 97.7%；25% 流量最少的测量区间命中率为 74.7%；25%流量最多区间为98.0%；99% 的 hypervisor 缓存命中的数据包速率低于 79,000 pkt/s

结论：缓存有效：命中率高；到用户态流量少

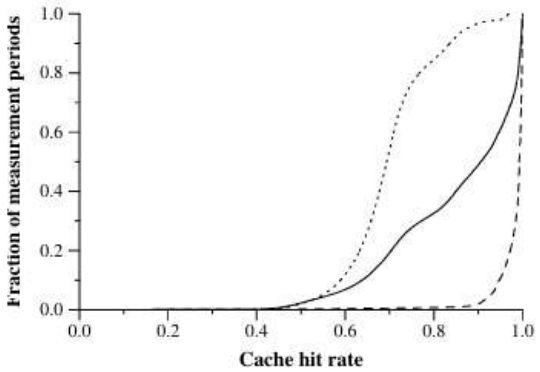


Figure 5: Hit rates during all (solid), busiest (dashed), and slowest (dotted) periods.

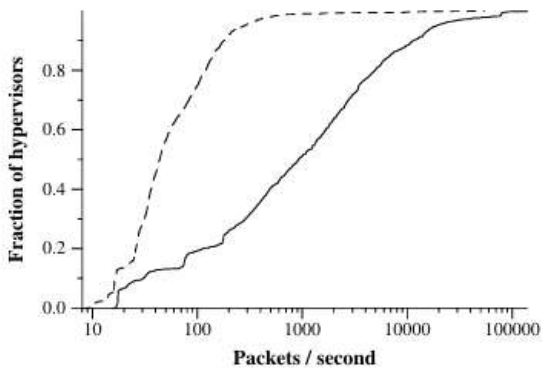


Figure 6: Cache hit (solid) and miss (dashed) packet counts.

CPU负载

数据：OVS用户态部分的CPU负载

观察：80% 的 hypervisor 上 ovs-vswitchd 的平均 CPU 使用率为 5% 或更低；超过 50% 的 hypervisor 使用率低于 2%。

结论：OVS不会带来高昂CPU的负载

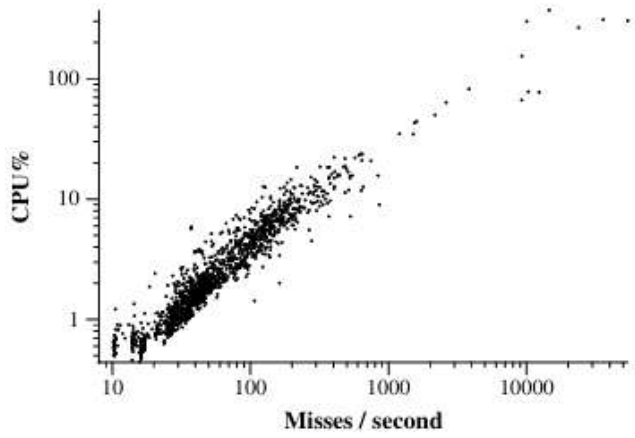


Figure 7: Userspace daemon CPU load as a function of misses/s entering userspace.

Performance in Production

微基准测试

使用以下流表测试OVS缓存感知的数据包分类算法的性能性能：

arp				(1)
ip	ip_dst=11.1.1.1/16			(2)
tcp	ip_dst=9.1.1.1	tcp_src=10	tcp_dst=10	(3)
ip	ip_dst=9.1.1.1/24			(4)

- **优先级排序** (Priority Sorting, 5.2)：匹配流#2的数据包可以省略TCP相关的字段，因为匹配过程中不会用到流#3的规则
- **分阶段查找** (Staged Lookup, 5.3)：目的IP地址不是9.1.1.1的数据包不用匹配TCP字段，因为这些数据包在IP地址匹配阶段就可以判定不匹配流#3
- **地址前缀追踪** (Address Prefix Tracking, 5.4)：可以忽略IP地址中的某些bit，尽管流#3有完整的IP地址

Performance in Production

各层缓存性能 (Cache layer performance)

实验环境：一台Linux服务器，配备2个8核心处理器和2个10Gb网卡。测试并发建立400个 Netperf 的TCP_CRR session：每个 session 不断建立一个 TCP 连接，发送一个字节的数据，关闭TCP连接。实验测量完成传输的速度 (tps)。

- **Baseline** (只有 **microflow caching**)：**37ktps**，并且需要~1百万条内核流，1 核的 CPU 时间
- **microflow + megaflow**：**120 ktps**
- **megaflow only**：**92 ktps**

Optimizations	ktps	Flows	Masks	CPU%
Megaflows disabled	37	1,051,884	1	45/ 40
No optimizations	56	905,758	3	37/ 40
Priority sorting only	57	794,124	4	39/ 45
Prefix tracking only	95	13	10	0/ 15
Staged lookup only	115	14	13	0/ 15
All optimizations	117	15	14	0/ 20

Table 1: Performance testing results for classifier optimizations. Each row reports the measured number of Netperf TCP_CRR transactions per second, in thousands, along with the number of kernel flows, kernel masks, and user and kernel CPU usage.

Microflows	Optimizations	ktps	Tuples/pkt	CPU%
Enabled	Enabled	120	1.68	0/ 20
Disabled	Enabled	92	3.21	0/ 18
Enabled	Disabled	56	1.29	38/ 40
Disabled	Disabled	56	2.45	40/ 42

Table 2: Effects of microflow cache. Each row reports the measured number of Netperf TCP_CRR transactions per second, in thousands, along with the average number of tuples searched by each packet and user and kernel CPU usage.

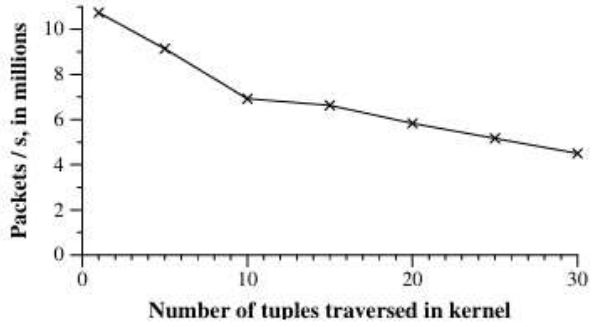


Figure 8: Forwarding rate in terms of the average number of megaflow tuples searched, with the microflow cache disabled.

Performance in Production

分类器优化收益 (Classifier optimization benefit)

测量了每一项优化单独带来的提升，以及所有优化同时启用时的整体提升。

结论：

- 每项优化都减少了测试中所需的 **kernel flows 数量**，进而降低了用户态所需的 **CPU 时间**
- 内核流表中的 **tuple（Masks）数量**增加，这提高了内核分类（kernel classification）的成本。但从**内核 CPU 时间减少**和 **TCP_CRR 吞吐量增加**可以看出，这些额外成本完全被 microflow 缓存和更少用户态往返的收益所抵消。
- **分类延迟**同样得到了降低

Optimizations	ktps	Flows	Masks	CPU%
Megaflows disabled	37	1,051,884	1	45/ 40
No optimizations	56	905,758	3	37/ 40
Priority sorting only	57	794,124	4	39/ 45
Prefix tracking only	95	13	10	0/ 15
Staged lookup only	115	14	13	0/ 15
All optimizations	117	15	14	0/ 20

Table 1: Performance testing results for classifier optimizations. Each row reports the measured number of Netperf TCP_CRR transactions per second, in thousands, along with the number of kernel flows, kernel masks, and user and kernel CPU usage.

Performance in Production

与内核交换机的对比(Comparison to in-kernel switch)

对比 OVS 和 Linux Bridge。

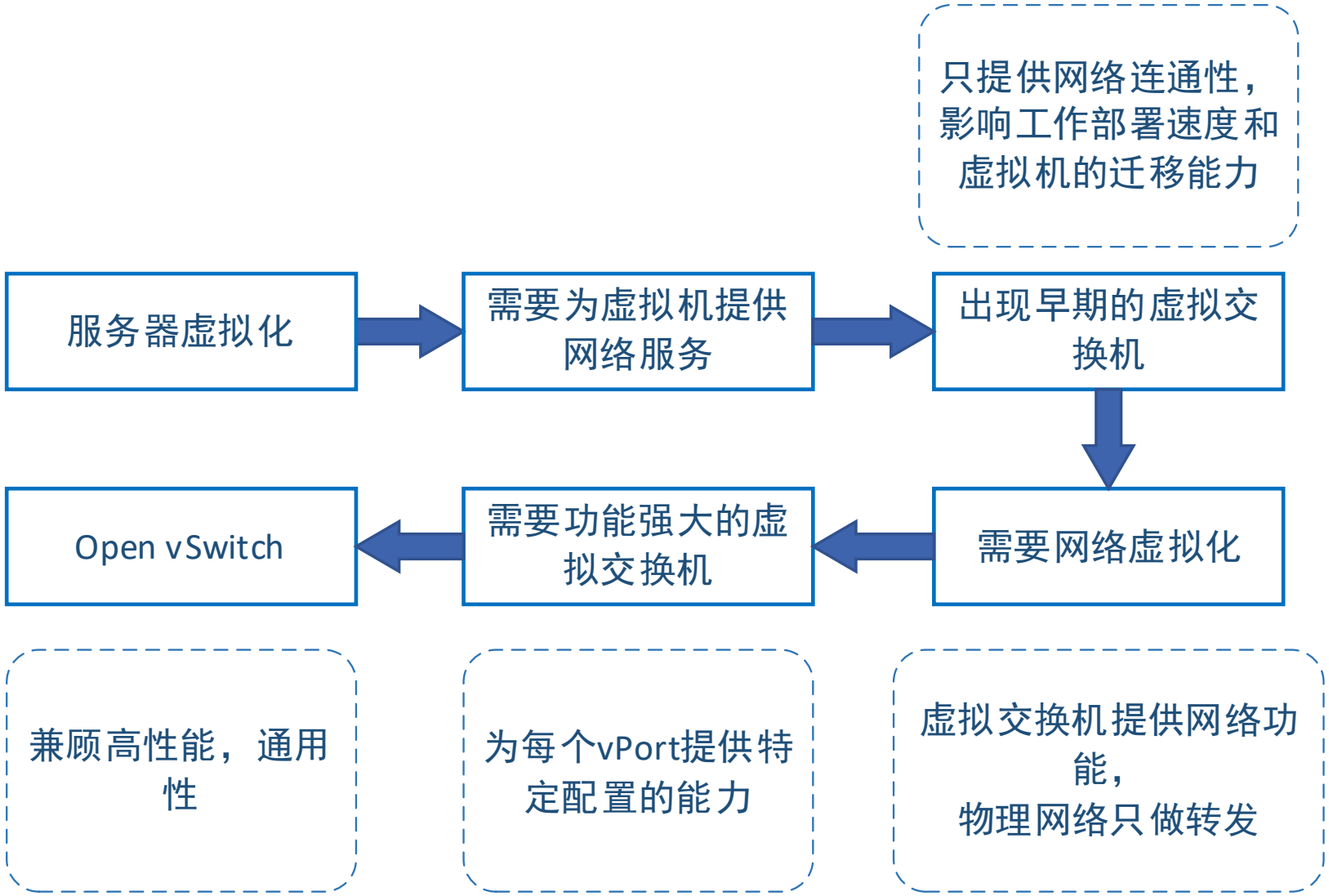
- 基本配置下
  - 两者**吞吐率** (18.8Gbps)、**TCP_CRR连接速率** (696 vs 688 ktps) 相近
  - OVS占用了更多的 **CPU** (161% vs 48%)
- Open vSwitch 添加一条“丢弃 STP BPDU 包”的流规则，并在 Linux bridge 上使用一条等效的 iptables 规则时：
  - **OVS** 的性能和 CPU 使用率保持不变
  - **Linux Bridge** 的连接速率下降到 512 ktps，CPU 使用率则飙升到 1279%

分析：Linux 内核的内置功能在每个数据包上都会产生固定开销，而 Open vSwitch 的开销通常是按 megaflow（而非按数据包）固定的。

Outline

I. Introduction
II. Design Constraints and Rationale
III. Design
IV. Flow Cache Design
V. Caching-aware Packet Classification
VI. Cache Invalidation
VII. Evaluation

Background


只提供网络连通性，影响工作部署速度和虚拟机的迁移能力
服务器虚拟化
需要为虚拟机提供网络服务
出现早期的虚拟交换机
需要网络虚拟化
需要功能强大的虚拟交换机
Open vSwitch
兼顾高性能，通用性
为每个vPort提供特定配置的能力
虚拟交换机提供网络功能，
物理网络只做转发

Design

分层架构：用户态（控制/分类）+ 内核态（快速路径）

双层缓存：microflow（精确匹配）+ megaflow（泛化匹配）

多种优化：分阶段匹配、优先级排序、前缀优化、分类器划分

高效更新：多线程、智能失效机制

Contribution

- 设计了适合虚拟化环境的可编程、高性能**虚拟交换架构**。
- 创造性地提出 megaflow + caching-aware classifier，使复杂 OpenFlow pipeline 可被高效折叠与缓存。
- 大规模**生产环境验证**：命中率约 98%，用户态 CPU 通常低于 5%。
- 推动虚拟交换机从 L2 功能扩展到支持多租户网络虚拟化的核心基础设施。