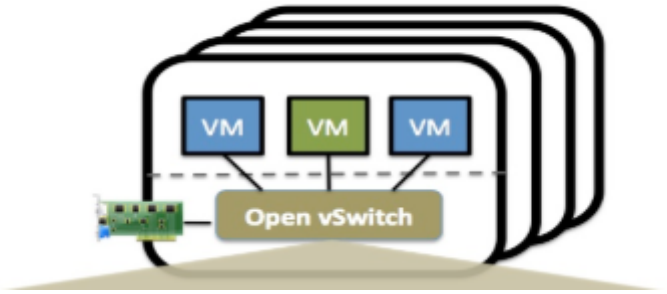
Open vSwitch中的流缓存设计（含源代码分析）

1. 什么是Open vSwitch?

在过去10几年里面，虚拟化已经改变了应用，数据，服务的实现部署方式。有研究表明，80%的x86工作负载已经虚拟化，其中大部分是虚拟机。

服务器的虚拟化给数据中心网络带来了根本性的变化。在传统的数据中心网络架构基础上，出现了一个新的，位于物理服务器内的接入层。这个新的接入层包含的设备是运行在x86服务器中的vSwitch，而这些vSwitch连接着一个服务器内的多个工作负载（包括容器和虚机）。



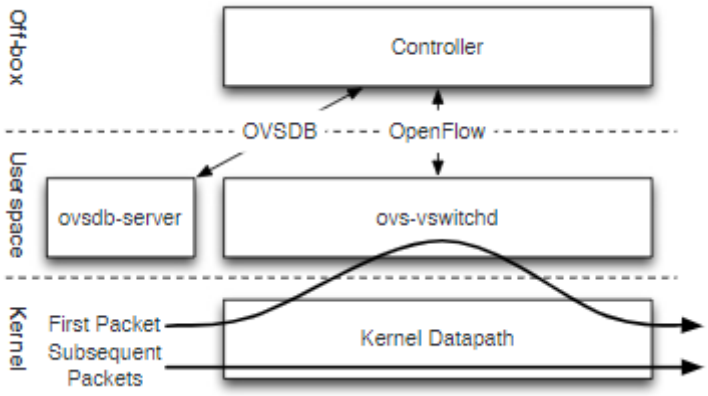
传统的交换机，不论是硬件的，还是软件的，所具备的功能都是预先内置的，需要使用某个功能的时候，进行相应的配置即可。而OpenVSwitch通过OpenFlow实现了交换机的可编程。OpenFlow可以定义网络包在交换机中的处理流程（pipeline），因此支持OpenFlow的交换机，其功能不再是固定的，通过OpenFlow可以软件定义OpenVSwitch所具备的功能。

2. 相关资料

官方文档：https://www.openvswitch.org/

源代码：https://github.com/openvswitch/ovs

3. Open vSwitch的设计架构



现在Open vSwitch主要由三个部分组成：

- ovsdb-server：OpenFlow本身被设计成网络数据包的一种处理流程，它没有考虑软件交换机的配置，例如配置QoS，关联SDN控制器等。ovsdb-server是Open vSwitch对于OpenFlow实现的补充，它作为OpenvSwitch的configuration database，保存Open vSwitch的持久化数据。
- ovs-vswitchd：运行在用户空间的转发程序，接收SDN控制器下发的OpenFlow规则。并且通知OVS内核模块该如何处理网络数据包。
- OVS内核模块：运行在内核空间的转发程序，根据ovs-vswitchd的指示，处理网络数据包

Open vSwitch中有快速路径（fast path）和慢速路径（slow path）。其中ovs-vswitchd代表了slow path，OVS内核模块代表了fast path。现在OpenFlow存储在slow path中，但是为了快速转发，网络包应该尽可能的在fast path中转发。因此，OpenVSwitch按照下面的逻辑完成转发。

当一个网络连接的第一个网络数据包（首包）被发出时，OVS内核模块会先收到这个数据包。但是内核模块现在还不知道如何处理这个包，因为所有的OpenFlow都存在ovs-vswitchd，因此它的默认行为是将这个包上送到ovs-vswitchd。ovs-vswitchd通过OpenFlow pipeline，处理完网络数据包送回给OVS内核模块，同时，ovs-vswitchd还会生成一串类似于OpenFlow Action，但是更简单的datapath action。这串datapath action会一起送到OVS内核模块。因为同一个网络连接的所有网络数据包特征（IP，MAC，端口号）都一样，当OVS内核模块收到其他网络包的时候，可以直接应用datapath action。

4. Open vSwitch中的流缓存设计

4.1 Microflow

在2007年，当在Linux上开始开发将成为Open vSwitch的代码时，只有内核数据包转发才能切实实现良好的性能，因此最初的实现将所有OpenFlow处理都放入了内核模块中。 该模块从NIC或VM接收到一个数据包，该数据包通过OpenFlow表进行分类（具有标准的OpenFlow匹配和操作），并在必要时对其进行修改，最后将其发送到另一个端口。由于在内核中开发和更新内核模块相对困难，因此该方法很快变得不切实际。

Open vSwitch采取的解决方案是将内核模块重新实现为microflow，其中单个缓存条目与OpenFlow支持的所有数据包头字段精确匹配。 通过将内核模块实现为简单的哈希表而不是复杂的通用数据包分类器，从而支持任意字段和掩码，从而实现了根本简化。 在这种设计中，缓存条目是非常细粒度的，并且与大多数单个传输连接的数据包都匹配。但是对于单个传输连接，即使是网络路径的更改以及IP TTL字段的更改也会导致未命中，并转移数据包发送到用户空间，用户空间将参考实际的OpenFlow流表来决定如何转发它。 这意味着影响性能的关键指标是流建立时间，即内核将缓存“未命中”报告给用户空间以及用户空间进行回复所花费的时间。

在多个Open vSwitch版本上，都采用了多种技术来减少microflow缓存的流建立时间。 例如，通过减少设置给定microflow所需的平均系统调用次数批量处理流的方法，总共可将性能提高约24％。 最终，我们还将流设置负载分配到了多个用户空间线程上，从而受益于多个CPU内核，等等。

尽管microflow缓存适用于大多数流量模式，但面对大量短连接时，其性能会严重下降。 在这种情况下，许多数据包会丢失高速缓存，不仅必须跨越内核-用户空间边界，而且还必须执行一系列昂贵的数据包分类。 尽管批处理和多线程可以减轻这种压力，但它们不足以完全支持这类工作负载。这要求Open vSwitch重新考虑流缓存设计。

4.2 Megaflow

为了避免这种性能极度恶化的情况，Open vSwitch 引入了 MegaFlow。和 MicroFlow 的精确匹配不同，MegaFlow 可以做到模糊匹配，一个条目可以匹配一组数据包。它的实现和用户态的 TSS 类似，但是在组织上有所不同。一是没有优先级，这样可以快速返回无需遍历所有的哈希表；二是 MegaFlow 中不像用户态中大量 table 组成了 pipeline，只通过一个 table 来进行匹配。

我们来看下面一条OpenFlow流表规则。

```
priority=200,ip,nw_dst=10.0.0.0/16 actions=output:1
```

它的作用就是匹配所有目的IP是10.0.0.0/16的网络包，从网卡1送出。对于下面的传输层连接，对应的action都是一样的。

```
11.0.0.2:5742 -> 10.0.0.10:3306
11.0.0.2:5743 -> 10.0.0.10:3306
11.0.0.2:5744 -> 10.0.0.11:3306
11.0.0.3:5742 -> 10.0.0.10:3306
```

但是对应于microflow cache来说，就要有4条cache，需要上送4次ovs-vswitchd。但是实际上，如果在kernel datapath如果有下面一条cache，只匹配目的IP地址：

```
ip,nw_dst=10.0.0.0/16 actions=output:1
```

那么数以亿计的传输层连接，可以在OVS内核模块，仅通过这条cache，完成转发。因为只有一条cache，所以也只有一次上送ovs-vswitchd。这样能大大减少内核态向用户态上送的次数，从而提升网络性能。这样一条非精确匹配的cache，被OpenVSwitch称为megaflow。

MegaFlow Cache 性能最关键的就是看如何能实现更好的泛化能力，即每个条目都能匹配尽可能多的数据包，减少用户态和内核态之间进行交互的次数。同时需要尽可能降低哈希查询的次数，在尽可能少的表里得到预期的结果。

4.3 Microflow vs Megaflow

Microflow的优点是，只有一个Hash table，一旦缓存建立，内核中只需要查找一次Hash table即可完成转发，缺点是每个不同的网络连接都需要一次内核空间上送用户空间，这有点多。Megaflow的优点是，合并了拥有相同的datapath action的网络连接所对应的缓存，对于所有这些网络连接，只需要一次内核空间上送用户空间，缺点是缓存建立以后，要查多次Hash table。如果每个Hash table命中的概率一样，那么平均需要查找 (n+1)/2 次hash table，最坏要查n个hash table。

4.4 现行方案

OVS内核模块采用Microflow+Megaflow的两级cache模式。Microflow作为第一级缓存，还是匹配所有OpenFlow可能的匹配的值，但是这时对应的Action不再是Datapath action，而是送到某个Megaflow的Hash table。当网络包送到kernel datapath时，会先在Microflow cache中查找，如果找到了，那么再接着送到相应的Megaflow hash table继续查找；如果没有找到，网络包会在kernel datapath的Megaflow cache中继续查找，如果找到了，会增加Microflow cache的记录，将后继类似的包直接指向相应的Megaflow Hash table；如果在Megaflow还是没有找到，那么就要上送ovs-vswitchd，通过OpenFlow pipeline生成Megaflow cache。

这样不论Megaflow cache有多少个hash table，对于长连接来说，只需要查找两次hash table即可完成转发，长连接的性能得到了保证。对于短连接来说，因为Megaflow cache的存在，也不用频繁的上送ovs-vswitchd，只需要在OVS内核模块多查几次hash table就行，短连接的性能也得到了保证。

5. 流缓存部分源码分析

```
//ovs-master/datapath/flow_table.h
...
/********************* microflow cache*************************************
*从mask_cache_entry结构体可以看出，microflow作为第一级缓存，将索引到上次使用的megaflow的
hash table
**************************************************************************/
struct mask_cache_entry {

  u32 skb_hash;         //flow标识，同一个flow有一致的hash值

  u32 mask_index;        //索引，指向megaflow的hash_table,对应代码中的mask[]

};
...
```

```
//ovs-master/datapath/flow_table.c
...
/*************************** megaflow cache*******************************
*flow_lookup结构体即完成在缓存中查找的工作
*这段代码展示了Open vSwitch的microflow+megaflow的缓存设计
*代码中涉及到rcu机制，是一种linux内存同步机制，不做过多讲解
**************************************************************************/
static struct sw_flow *flow_lookup(struct flow_table *tbl,
                   struct table_instance *ti,
                   const struct mask_array *ma,
                   const struct sw_flow_key *key,
                   u32 *n_mask_hit,
                   u32 *index)
{
    struct sw_flow_mask *mask;  //之后单独分析此结构体
    struct sw_flow *flow;
    int i;
//快速路径，根据索引值，直接查找对应hash_table
    if (*index < ma->max) {                       //合法性检查，养成好习惯
        mask = rcu_dereference_ovsl(ma->masks[*index]);
        if (mask) {
            flow = masked_flow_lookup(ti, key, mask, n_mask_hit);//在对应的hash
table中直接进行查找，之后单独分析此函数
            if (flow)
                return flow;
        }
    }
//快速查找未命中，那么就要遍历mask[]数组，即遍历megaflow的hash table
    for (i = 0; i < ma->max; i++)  {
//因为对于索引值对应的hash table已经查找过，未找到，所以直接跳过，避免重复查找
        if (i == *index)
            continue;

        mask = rcu_dereference_ovsl(ma->masks[i]);
```

```
        if (!mask)
            continue;

        flow = masked_flow_lookup(ti, key, mask, n_mask_hit);
        if (flow) {                         //最终找到了
            *index = i;                     //更新索引值，返回
            return flow;
        }
    }

    return NULL;            //最终未找到，之后需要上传到用户态处理
}
...
```

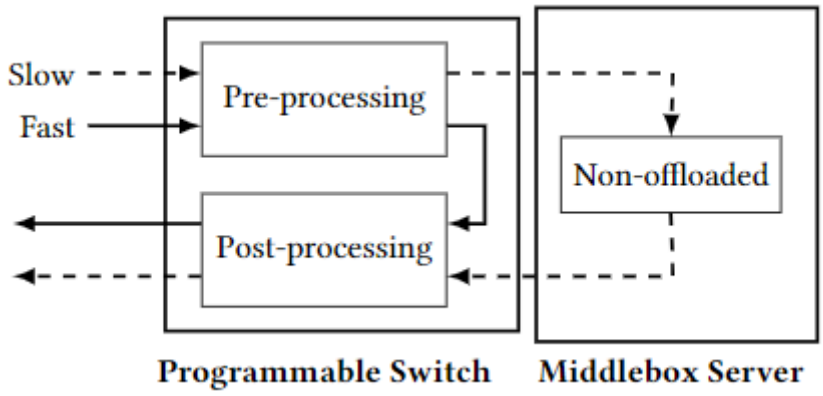
```
//ovs-master/datapath/flow.h
...
/*****************************************************************************
*mask[]对应的结构体，hash table的key值就存在这个结构体中
*****************************************************************************/
struct sw_flow_mask {
    int ref_count;
    struct rcu_head rcu;
    struct sw_flow_key_range range;
    struct sw_flow_key key;
};
...
```

```
//ovs-master/datapath/flow.h
...
/*****************************************************************************
*masked_flow_lookup()对应的结构体，hash table的value值就存在这个结构体中
*****************************************************************************/
struct sw_flow {
    struct rcu_head rcu;
    struct {
        struct hlist_node node[2];
        u32 hash;
    } flow_table, ufid_table;
    int stats_last_writer;

    struct sw_flow_key key;
    struct sw_flow_id id;
    struct cpumask cpu_used_mask;
    struct sw_flow_mask *mask;
    struct sw_flow_actions __rcu *sf_acts; //hash table的value值即对应的执行动作，存在sw_flow_actions结构体之中
    struct sw_flow_stats __rcu *stats[];
};
...
```

6. 引申

在sigcomm2020《 Gallium: Automated Software Middlebox Offloading to Programmable Switches》论文中，涉及到类似的问题。



comparison between Gallium and FastClick. We see that the reduction in flow completion time is concentrated on the long flows. This behavior is because long flows will have the majority of their packets handled by the programmable switch instead of the server.

同样面对大量短流表现不佳的问题，其实就可以借鉴Open vSwitch的这种思路。

参考文献

[1] Open vSwitch. https://www.openvswitch.org/

[2] Pfaff B, Pettit J, Koponen T, et al. The design and implementation of open vswitch[C]//12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15). 2015: 117-130.

[3] Zhang K, Zhuo D, Krishnamurthy A. Gallium: Automated Software Middlebox Offloading to Programmable Switches[C]//Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication. 2020: 283-295.