* **每个数据包查找datapath中流表时间**

解释：统计每一个数据包经过datapath中查找内核态流表的时间。每一个到达OVS的数据包都需要先查询内核态流表进行确定如何处理当前数据包，比如转发、丢弃。由于内核流表采用了缓存的设计思想，与刚刚经过OVS数据包相似的数据包会获得比较快的查表速度。同时内核流表的大小也在一定程度上影响查表速度。因此，该特征可以反应当前datapath的拥塞状况,同时也能反应流的变化频率和内核流表的状态。

英文：

Name：Time for each packet to look up the flow table in the datapath

Description：record the time for each packet to look up the flow table in the datapath,it can reflect the congestion status of the current datapath,and also can reflect the variation of the IP of the packet.

相关实现：

|  |
| --- |
| /\* Look up flow. \*/  struct timeval start\_find,end\_find;  do\_gettimeofday(&start\_find);  flow = ovs\_flow\_tbl\_lookup\_stats(&dp->table, key, skb\_get\_hash(skb),  &n\_mask\_hit);//查询内核态流表函数  do\_gettimeofday(&end\_find); |

代码逻辑：ovs\_flow\_tbl\_lookup\_stats函数为内核态流表查询函数，在datapath.c中的ovs\_dp\_process\_packet函数中被调用。只需要统计调用开始时间和结束时间

* **命中内核流表的数据包数目**

解释：统计经过datapath的数据包中命中内核态流表的数目。每一个到达OVS的数据包都需要先查询内核态流表进行确定如何处理当前数据包。对于命中了内核流表的数据包来说，数据包无需上传到用户态，避免了一次数据包拷贝以及许多系统调用，可以直接在内核态下对数据包进行处理，大大加快了数据包的处理速度。该特征可以反应出内核态流表的命中情况，转发给用户态数据包的数量，OVS的工作状况。

英文：

Name: Number of packets hitting the flow table in the datapath

Description:record the number of packets hitting the flow table in the datapath.It can reflect how many packets had hit the flow table in the datapath,and reflect how many packets had been transmit to user space,and roughly reflect how many flows in the network;

相关实现：

|  |
| --- |
| flow = ovs\_flow\_tbl\_lookup\_stats(&dp->table, key, skb\_get\_hash(skb),  &n\_mask\_hit);//查询内核态流表函数  do\_gettimeofday(&end\_find);  if(flow){//如果查找到相应的流表项  atomic\_inc(&hit\_kernal\_table);  } |

代码逻辑：ovs\_flow\_tbl\_lookup\_stats函数会返回查找到的流表项，如果没有查找到则返回NULL；可以通过返回值判断是否命中流表

* **命中内核mask\_cache哈希表的次数**

解释：经过OVS内核的数据包都要进行内核流表查找。查找内核流表过程中，必需要先找到数据包对应个mask。mask能用于模糊匹配，能让内核流表容错率更高。该特征统计在内核流表查询过程中，对于mask\_cache哈希表（存储着mask，mask用于模糊匹配）查询过程中的命中次数。该哈希表使用skb\_hash（由linux内核为skb\_buff生成的hash值，可以唯一表示skb\_buff）的低八位进行哈希。如果发生冲突，那么就使用skb\_hash的下一个低八位进行哈希。该特征能够反映出经过OVS数据包的四元组变化情况。

相关实现：

|  |
| --- |
| if (e->skb\_hash == skb\_hash) {  atomic\_inc(&hit\_mask\_cache);  flow = flow\_lookup(tbl, ti, ma, key, n\_mask\_hit,  &e->mask\_index);  if (!flow)  e->skb\_hash = 0;  return flow;  } |

代码逻辑：添加的代码位于flow\_table.c的ovs\_flow\_tbl\_lookup\_stats函数中，只需要在命中哈希表的条件中自增相应的统计量即可。

* **查询内核mask\_cache哈希表的次数**

解释：经过OVS内核的数据包都要进行内核流表查找。查找内核流表过程中，必需要先找到数据包对应个mask。mask能用于模糊匹配，能让内核流表容错率更高。该特征统计在内核流表查询过程中，对于mask\_cache哈希表（存储着mask，mask用于模糊匹配）查询次数，该哈希表使用skb\_hash（由linux内核为skb\_buff生成的hash值，可以唯一表示skb\_buff）的低八位进行哈希。如果发生冲突，那么就使用skb\_hash的下一个低八位进行哈希。该特征能够反映出经过OVS数据包的四元组变化情况。

相关实现：

|  |
| --- |
| for (seg = 0; seg < MC\_HASH\_SEGS; seg++) {  int index = hash & (MC\_HASH\_ENTRIES - 1);  struct mask\_cache\_entry \*e;  atomic\_inc(&total\_times\_mask\_cache);  e = &entries[index]; |

代码逻辑：添加的代码位于flow\_table.c的ovs\_flow\_tbl\_lookup\_stats函数中，只需要在查询哈希表位置中自增相应的统计量即可。

* **查询内核流表hash表的次数**

解释：每个经过OVS内核的数据包都要进行内核流表查找。在内核流表查询过程中先找到数据包的mask。然后将mask与数据包的四元组进行与操作，获得的结果进行一定处理，然后作为内核流表hash表（存储着流表项）的索引。该特征统计在内核流表查询过程中，对于内核流表hash表查询次数。该哈希表使用链式结构处理冲突。该特征能够反映出经过OVS的流的变化情况。

相关实现：

|  |
| --- |
| ovs\_flow\_mask\_key(&masked\_key, unmasked, false, mask);// 进行与操作  hash = flow\_hash(&masked\_key, &mask->range);  head = find\_bucket(ti, hash);//查表  (\*n\_mask\_hit)++;  atomic\_inc(&total\_times\_hash\_flow);//自增 |

代码逻辑：添加的代码位于flow\_table.c的masked\_flow\_lookup函数中，只需要在查询哈希表位置中自增相应的统计量即可。

* **命中内核流表hash表的次数**

解释：每个经过OVS内核的数据包都要进行内核流表查找。在内核流表查询过程中先找到数据包的mask。然后将mask与数据包的四元组进行与操作，获得的结果进行一定处理，然后作为内核流表hash表（存储着流表项）的索引。该特征统计在内核流表查询过程中，对于内核流表hash表的命中次数。该哈希表使用链式结构处理冲突。该特征能够反映出经过OVS的流的变化情况。

相关实现：

|  |
| --- |
| hlist\_for\_each\_entry\_rcu(flow, head, flow\_table.node[ti->node\_ver]) {//遍历链表中的流表项  if (flow->mask == mask && flow->flow\_table.hash == hash &&  flow\_cmp\_masked\_key(flow, &masked\_key, &mask->range)){  atomic\_inc(&hit\_hash\_flow);  return flow;  }  } |

代码逻辑：添加的代码位于flow\_table.c的masked\_flow\_lookup函数中，只需要在判断哈希表命中位置中自增相应的统计量即可。

* **ovs\_flow\_cmd\_set()、ovs\_flow\_cmd\_get()、ovs\_flow\_cmd\_del()执行次数**

解释：这三个函数是控制器对ovs内核的流表的控制，可以反应控制器对内核流表的控制情况,以及流表的变化。OVS用户态程序通过netlink向内核态发送的与控制流表相关的命令，比如流表项的修改、删除等，这些命令最终会调用Datapath中的这三个函数。ovs\_flow\_cmd\_set()可以修改相应的内核流表项，ovs\_flow\_cmd\_get()将相应的内核流表项发送到用户态，ovs\_flow\_cmd\_del()可以删除相应的内核流表项。

英文：

Name:Frequancy of **ovs\_flow\_cmd\_set(),**Frequancy of **ovs\_flow\_cmd\_get(),**Frequancy of **ovs\_flow\_cmd\_del()**

Description: Frequancy of **ovs\_flow\_cmd\_set() can reflect the degree of control from Controller,and reflect the change of the flow table.**

相关实现：

|  |
| --- |
| static int ovs\_flow\_cmd\_set(struct sk\_buff \*skb, struct genl\_info \*info)  {  atomic\_inc(&cmd\_set\_ex\_times);//原子操作，自增，统计该函数执行次数  ……余下略  } |

代码逻辑：在相应函数使用原子操作来进行统计。

* **ovs\_execute\_actions()执行次数**

解释：当数据包进入内核态查询流表命中时，需要执行相应的数据包处理，比如丢弃、转发，而ovs\_execute\_actions()函数就是用于使用特定动作去处理数据包。该统计值可以反应ovs中处理数据包的数目。

英文：

Name:Frequency of **ovs\_execute\_actions()**

Description: Accouring to the serach result on the datapath flow table, **ovs\_execute\_actions() will do corresponding actions.This feature reflect how many packets had been processed.**

相关实现：

|  |
| --- |
| int ovs\_execute\_actions(struct datapath \*dp, struct sk\_buff \*skb,  const struct sw\_flow\_actions \*acts,  struct sw\_flow\_key \*key)  {  atomic\_inc(&ovs\_execute\_actions\_times);//原子操作自增  ……余下略  } |

代码逻辑：在相应函数使用原子操作来进行统计。

* **upcall的数目**

解释：对于每一个经过OVS的数据包都需要查找内核流表以确定处理数据包的方法，但是有时候内核流表会不命中，这个时候为了找到数据包的处理方法，就要向OVS用户态发送一个upcall，该upcall包含了未命中内核流表的数据包，OVS用户态会到用户态的流表中查找相应的流表项来对数据包进行处理。该统计可以反应数据包中查询内核态流表失败的次数，可以间接的反应packet\_in的数量。

英文：

Name:Number of Upcall

Description:This feature records the number of packet which can’t find in the datapath flow table and then was sent to user space.It can reflect how many packets were fail to find in the datapath flow table,and reflect how many packet\_ins were sent to the Controller.

相关实现：

|  |
| --- |
| atomic\_inc(&upcall\_nummber);//原子操作自增  err = genlmsg\_unicast(ovs\_dp\_get\_net(dp), user\_skb, upcall\_info->portid);//upcall发送函数 |

代码逻辑：在upcall发送函数genlmsg\_unicast之前添加原子操作自增。

* **upcall延时**

解释：统计每一个要从内核态发送到用户态的数据包所需要的时间。由于数据包沒有命中内核流表，OVS内核的Datapath模块需要将数据包通过upcall发送到用户态。Upall是通过linux中的Generic Netlink来发送到用户态。该统计可以反应，当前主机的性能，同时也可以反应网络的拥塞情况，用户态和内核态信息的交互。

英文：

Name:Upcall delay

Description:This feature records how much time a packet which can’t find in datapath flow table was sent to user space.It can reflect the status of the load of OVS and network situation from the point of view of whether crowded.If the Upcall delay is very large,it means that there are too many packets waiting to process or the datapath flow table match the packet badly.

相关实现：修改netlink通信中包头部大小（在每一个netlink数据包后面添加一个起始时间）

|  |
| --- |
| static size\_t upcall\_msg\_size(const struct dp\_upcall\_info \*upcall\_info,  unsigned int hdrlen, int actions\_attrlen)  {  struct timeval k;  size\_t size = NLMSG\_ALIGN(sizeof(struct ovs\_header))  + nla\_total\_size(hdrlen) /\* OVS\_PACKET\_ATTR\_PACKET \*/  + nla\_total\_size(ovs\_key\_attr\_size()) /\* OVS\_PACKET\_ATTR\_KEY \*/  + nla\_total\_size(sizeof(unsigned int)) /\* OVS\_PACKET\_ATTR\_LEN \*/  + nla\_total\_size(sizeof(k)); /\*OVS\_PACKET\_ATTR\_JIFFIES\*/ |

添加相应的关键字

|  |
| --- |
| static const struct nla\_policy packet\_policy[OVS\_PACKET\_ATTR\_MAX + 1] = {  [OVS\_PACKET\_ATTR\_PACKET] = { .len = ETH\_HLEN },  [OVS\_PACKET\_ATTR\_JIFFIES] = {.type = NLA\_UNSPEC},//timeval结构体  [OVS\_PACKET\_ATTR\_KEY] = { .type = NLA\_NESTED },  [OVS\_PACKET\_ATTR\_ACTIONS] = { .type = NLA\_NESTED },  [OVS\_PACKET\_ATTR\_PROBE] = { .type = NLA\_FLAG },  [OVS\_PACKET\_ATTR\_MRU] = { .type = NLA\_U16 },  }; |

|  |
| --- |
| enum ovs\_packet\_attr {  OVS\_PACKET\_ATTR\_UNSPEC,  OVS\_PACKET\_ATTR\_PACKET, /\* Packet data. \*/  OVS\_PACKET\_ATTR\_KEY, /\* Nested OVS\_KEY\_ATTR\_\* attributes. \*/  OVS\_PACKET\_ATTR\_ACTIONS, /\* Nested OVS\_ACTION\_ATTR\_\* attributes. \*/  OVS\_PACKET\_ATTR\_USERDATA, /\* OVS\_ACTION\_ATTR\_USERSPACE arg. \*/  OVS\_PACKET\_ATTR\_JIFFIES, /\*OVS\_PACKET\_ATTR\_JIFFIES gary's code\*/  OVS\_PACKET\_ATTR\_EGRESS\_TUN\_KEY, /\* Nested OVS\_TUNNEL\_KEY\_ATTR\_\*  attributes. \*/  OVS\_PACKET\_ATTR\_UNUSED1,  OVS\_PACKET\_ATTR\_UNUSED2,  OVS\_PACKET\_ATTR\_PROBE, /\* Packet operation is a feature probe,  error logging should be suppressed. \*/  OVS\_PACKET\_ATTR\_MRU, /\* Maximum received IP fragment size. \*/  OVS\_PACKET\_ATTR\_LEN, /\* Packet size before truncation. \*/  \_\_OVS\_PACKET\_ATTR\_MAX  }; |

|  |
| --- |
| parse\_odp\_packet(const struct dpif\_netlink \*dpif, struct ofpbuf \*buf,  struct dpif\_upcall \*upcall, int \*dp\_ifindex)  {  static const struct nl\_policy ovs\_packet\_policy[] = {  /\* Always present. \*/  [OVS\_PACKET\_ATTR\_PACKET] = { .type = NL\_A\_UNSPEC,  .min\_len = ETH\_HEADER\_LEN },  [OVS\_PACKET\_ATTR\_KEY] = { .type = NL\_A\_NESTED },  [OVS\_PACKET\_ATTR\_JIFFIES] = {.type = NL\_A\_UNSPEC },//用户态下的  /\* OVS\_PACKET\_CMD\_ACTION only. \*/  [OVS\_PACKET\_ATTR\_USERDATA] = { .type = NL\_A\_UNSPEC, .optional = true },  [OVS\_PACKET\_ATTR\_EGRESS\_TUN\_KEY] = { .type = NL\_A\_NESTED, .optional = true },  [OVS\_PACKET\_ATTR\_ACTIONS] = { .type = NL\_A\_NESTED, .optional = true },  [OVS\_PACKET\_ATTR\_MRU] = { .type = NL\_A\_U16, .optional = true }  };  …略 |

内核态下往数据包输入数据（在queue\_userspace\_packet函数中）

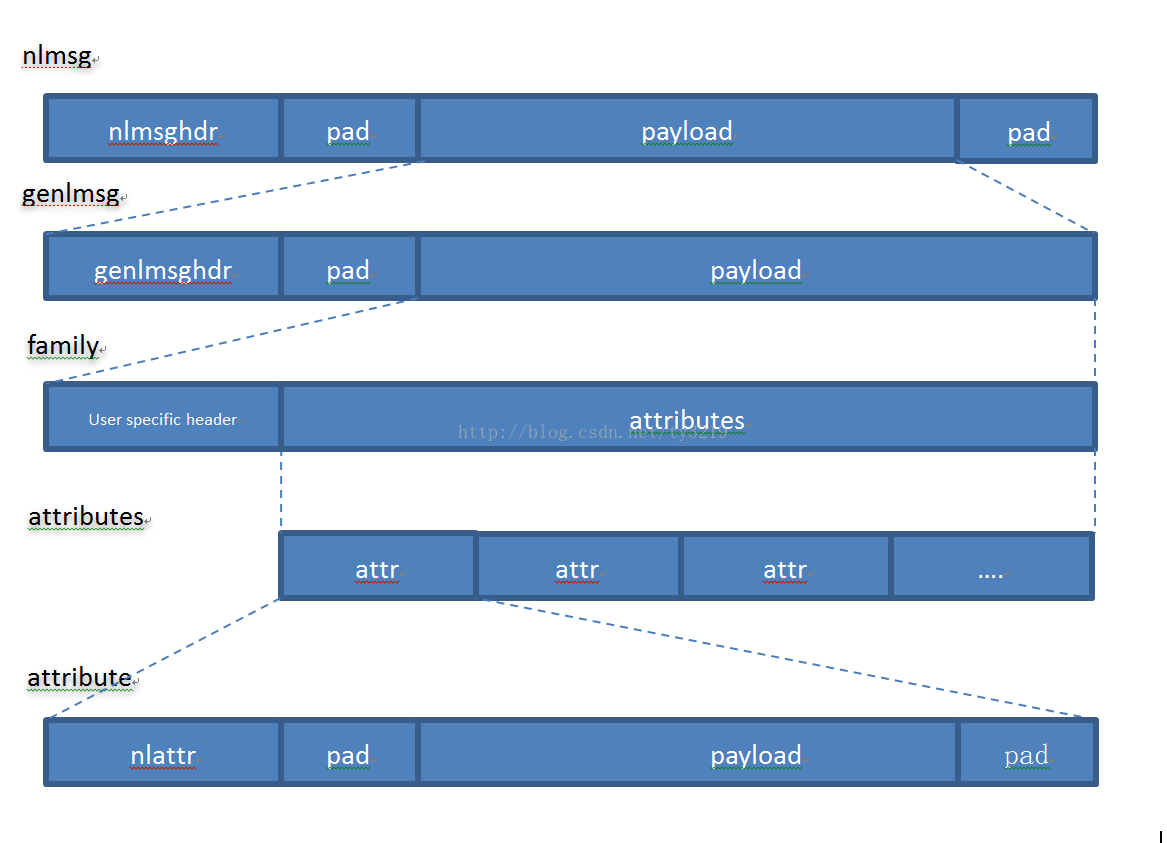
|  |
| --- |
| if (upcall\_info->userdata)  \_\_nla\_put(user\_skb, OVS\_PACKET\_ATTR\_USERDATA,  nla\_len(upcall\_info->userdata),  nla\_data(upcall\_info->userdata));//以此行为参照  \_\_nla\_put(user\_skb,OVS\_PACKET\_ATTR\_JIFFIES,sizeof(txc),&txc);//添加数据 |

在用户态取出数据

|  |
| --- |
| upcall->type = type;  upcall->key = CONST\_CAST(struct nlattr \*,  nl\_attr\_get(a[OVS\_PACKET\_ATTR\_KEY]));  upcall->key\_len = nl\_attr\_get\_size(a[OVS\_PACKET\_ATTR\_KEY]);  dpif\_flow\_hash(&dpif->dpif, upcall->key, upcall->key\_len, &upcall->ufid);  upcall->userdata = a[OVS\_PACKET\_ATTR\_USERDATA];  //取数据参照上面代码，可得如下  struct timeval time1,time2;  time1=\*(struct timeval\*)nl\_attr\_get(a[OVS\_PACKET\_ATTR\_JIFFIES]);//获取在内核态添加的起始时间  gettimeofday(&time2, NULL);  openlog("info",LOG\_PID,LOG\_LOCAL4);  unsigned int used\_time=(time2.tv\_sec-time1.tv\_sec)\*1000000+(time2.tv\_usec-time1.tv\_usec);//根据当前时间计算时延  syslog(LOG\_DEBUG,"gary\_upcall:%llu %lu",time2.tv\_sec,used\_time);//输出到日志 |

代码逻辑：

genl netlink数据包格式如下：



数据包分了4层，用户的实际数据封装在attribute里，一个或多个attribute可以被封装在用户自定义的一个family报文里。upcall中attribute包含了数据包、key值等attribute。

在每个发送给用户态的数据包中添加一个时间戳的attribute需要修改一下四个地方

1. 修改包大小
2. 在内核添加该ATTR的类型 （在用户态中openvswitch.h和内核态中的datapatch.c中添加）
3. 在datapach.c中queue\_userspace\_packet函数中向要发送的数据包中添加jiffies变量
4. 在用户态接收端函数parse\_odp\_packet处添加接收内核发送上来的jiffies变量

* **upcall消息长度**

解释: 统计从内核态发给用户态的upcall数据包的长度。该统计值反映了每一个数据包的大小。

英文：

Name:The length of Upcall message

Description:This feature records the length of Upcall message.It can reflect the size of the packet.

相关实现：

|  |
| --- |
| len = upcall\_msg\_size(upcall\_info, hlen - cutlen,  OVS\_CB(skb)->acts\_origlen); |

代码逻辑：该统计值ovs源码datapath.c中的queue\_userspace\_packet函数中有，可以直接输出

* **upcall消息发送失败的次数（数据全0）**

解释：统计从内核态发给用户态的upcall数据包中发送失败的次数。该统计值可以反应出当前网络的状况，如果失败次数过多，说明ovs达到性能瓶颈。

相关实现：

|  |
| --- |
| err = genlmsg\_unicast(ovs\_dp\_get\_net(dp), user\_skb, upcall\_info->portid);//upcall发送函数  user\_skb = NULL;  out:  if (err){//如果发送失败  skb\_tx\_error(skb);  atomic\_inc(&upcall\_fail);  } |

代码逻辑：genlmsg\_unicast函数为发送函数，该函数会有一个返回值，用以表明发送是否失败

* **接收到控制器的数据包数目**

解释：统计ovs接收到控制器数据包的数目，该统计值可以反应ovs于控制器的交互情况，同时反应控制器对ovs的控制频繁程度。

英语：

Name: Number of packets receiving from the Controller

Description:The feature records the number of packets receiving from the Controller. Only the packet which is fail to find in the flow table is sent to the Controller.

相关实现：

|  |
| --- |
| struct ofpbuf \*of\_msg = rconn\_recv(ofconn->rconn);//接收函数  if (!of\_msg) {  break;  }  if (mgr->fail\_open) {  fail\_open\_maybe\_recover(mgr->fail\_open);  }  //gary code  \_\_sync\_fetch\_and\_add (&count\_recv , 1 ) ;//原子操作，自增  //gary code end |

代码逻辑：在接收函数ronn\_recv后面添加相应的统计变量

* **发送给控制器的数据包数目**

解释：统计ovs发送到控制器数据包的数目，该统计值可以反应ovs于控制器的交互情况，同时可以大致反应packet\_in消息的数目。

Name: Number of packets sending to the Controller

Description:The feature records the number of packets sending to the Controller. Only the packet which is fail to find in the flow table is sent to the Controller.

相关实现：

|  |
| --- |
| int count\_send = 0;  static void  ofconn\_send(const struct ofconn \*ofconn, struct ofpbuf \*msg,  struct rconn\_packet\_counter \*counter)  {  //gary code  \_\_sync\_fetch\_and\_add (&count\_send , 1 ) ;//原子操作，自增  //gary code end  ofpmsg\_update\_length(msg);  rconn\_send(ofconn->rconn, msg, counter);  } |

代码逻辑：在发送函数ofconn\_send中添加相应的统计变量

* **ovs-vswitch.c中main函数执行循环的次数**

解释：统计用户态中main函数执行的个数。mian函数中包含了一个循环来处理数据还有有相应用户的输入。该统计能够反应ovs目前的性能状态。

相关实现：

|  |
| --- |
| while (!exiting) {  main\_times++;//自增，无需考虑多线程  memory\_run();  if (memory\_should\_report()) {  struct simap usage;  simap\_init(&usage);  bridge\_get\_memory\_usage(&usage);  memory\_report(&usage);  simap\_destroy(&usage);  }  bridge\_run();  unixctl\_server\_run(unixctl);  netdev\_run();  memory\_wait();  bridge\_wait();  unixctl\_server\_wait(unixctl);  netdev\_wait();  if (exiting) {  poll\_immediate\_wake();  }  poll\_block();  if (should\_service\_stop()) {  exiting = true;  }  } |

代码逻辑：在循环中添加特定的代码统计；

* **udpif\_upcall\_handler()函数中循环执行的次数**

解释：当OVS内核发送upcall到用户态时，需要用户态程序进行接收和处理。udpif\_upcall\_handler()函数为用户态为了处理upcall新开的线程,用来处理从内核态发送上来的upcall的消息。统计该函数中循环的次数可以反应出用户态接收到upcall消息的数目。

相关实现：

|  |
| --- |
| int count\_udpif\_upcall\_handler=0;  static void \*  udpif\_upcall\_handler(void \*arg)  {  struct handler \*handler = arg;  struct udpif \*udpif = handler->udpif;  while (!latch\_is\_set(&handler->udpif->exit\_latch)) {  if (recv\_upcalls(handler)) {  poll\_immediate\_wake();  } else {  dpif\_recv\_wait(udpif->dpif, handler->handler\_id);  latch\_wait(&udpif->exit\_latch);  }  poll\_block();  \_\_sync\_fetch\_and\_add(&count\_udpif\_upcall\_handler,1);//原子操作，自增  }  return NULL;  } |

代码逻辑：在循环中添加相应的统计；

* **每个数据包查找用户态流表时间**

解释：当数据包未能命中OVS内核流表时，需要将数据包发送到OVS用户态进行处理。OVS用户态程序拥有完整的OVS流表，OVS内核流表只是其一部分。通过查询用户态流表，可以确定对数据包的处理方式。该特征统计每一个从内核态发送给用户态的数据包查找用户态流表的时间。该统计可以反应用户态流表的命中情况。

相关实现：

|  |
| --- |
| struct timeval start ,end; //gary code  gettimeofday(&start,NULL);  xerr = xlate\_actions(&xin, &upcall->xout);//该函数会执行查找流表的操作  gettimeofday(&end,NULL);  int user\_table\_time=(int)end.tv\_usec-(int)start.tv\_usec;  user\_table\_time=1000000\*((int)end.tv\_sec-(int)start.tv\_sec)+user\_table\_time;  openlog("debug",LOG\_PID,LOG\_LOCAL4);  syslog(LOG\_DEBUG,"user\_table\_time:%llu %d",end.tv\_sec,user\_table\_time); |

代码逻辑：xlate\_action函数会执行查找流表的操作，因此，只要获取该函数执行之前以及之后的时间就可以计算出查找时间。

* **用户态流表数目**

解释：统计当前时刻用户态中流表的数目，该数目可以反应控制器下放给ovs的流表数目。

相关实现：

|  |
| --- |
| unsigned int count\_user\_table=0;  static struct rule\_dpif \*  rule\_dpif\_lookup\_in\_table(struct ofproto\_dpif \*ofproto, ovs\_version\_t version,  uint8\_t table\_id, struct flow \*flow,  struct flow\_wildcards \*wc)  {  struct classifier \*cls = &ofproto->up.tables[table\_id].cls;  struct timeval time1;  gettimeofday(&time1, NULL);  count\_user\_table=0;  for(int i=0;i<ofproto->up.n\_tables;i++){  count\_user\_table+=ofproto->up.tables[i].cls.n\_rules;  }  return rule\_dpif\_cast(rule\_from\_cls\_rule(classifier\_lookup(cls, version,  flow, wc)));  } |

代码逻辑：up.n\_tables记录拥有个流表个数，ofproto->up.tables为流表数组首地址，遍历每一个流表中所拥有的流表数目即可

* **命中用户态流表的数目**

解释：数据包在到达OVS内核时，由于未能命中内核态流表，需要发送到用户态查询用户态的流表，以确定如何处理数据包。该特征统计从内核态转发到用户态的数据包中，能够命中用户态流表的个数。该统计可以反应用户态流表的命中率，以及当前网络中流的数目。

相关实现：

|  |
| --- |
| if (rule) {//当命中用户态流表时  \_\_sync\_fetch\_and\_add (&hit\_user\_table, 1 ) ;//原子操作，自增  goto out; /\* Match. \*/ |

代码逻辑：查询流表函数rule\_dpif\_lookup\_from\_table中有查询命中的判断，只需要在相应的判定添加统计量即可。

* **执行内核态命令失败的次数**

解释：OVS用户态在处理由内核发送上来的数据包时，通过查找用户态流表确定了如何处理数据包，然后通过向内核态发送相应的命令去进行处理。该统计从用户态发给内核态命令执行失败的次数。该数据可以反应出控制器对于当前ovs的调度情况，同时还能反应出当前流经ovs的数据量。

相关实现：

|  |
| --- |
| if(err!=0){  atomic\_inc(&cmd\_fail\_times);//原子操作自增  } |

代码逻辑：首先用户态命令是通过netlink机制发送到内核态下。netlink机制中内核态接收到命令消息后就会去执行响应函数ovs\_packet\_cmd\_execute。该函数会执行相应的命令，对进行结果返回值进行判断即可知道是否执行成功。