清华大学

SDN 流表查询中的 cache 加速技术调查报告

2018-11-9

小组成员及分工

姓名	学号	工作内容
张豪	2017213815	阅读文献《FlowShadow Keeping Update Consistency in Software-based OpenFlow Switches》、《Enhancement of Flow in SDN by Embedded Cache》，整合文档和 ppt。
崔浩	2018214160	阅读文献《FDRC Flow-driven rule caching optimization in software defined networking 》、《 Infinite CacheFlow in Software-Defined Networks》, 编写相应文档和 ppt
钟春蒙	2018214147	阅读文献《An Efficient Flow Cache algorithm with Improved Fairness in Software-Defined Data Center Network》、《The Design and Implementation of Open vSwitch》，编写相应文档和 ppt
张苏坤	2018214135	阅读文献《An Efficient Flow Cache algorithm with Improved Fairness in Software-Defined Data Center Network_wrapper 》、《 CacheFlow: Dependency-Aware Rule-Caching for Software-Defined Networks》, 编写相应文档和 ppt

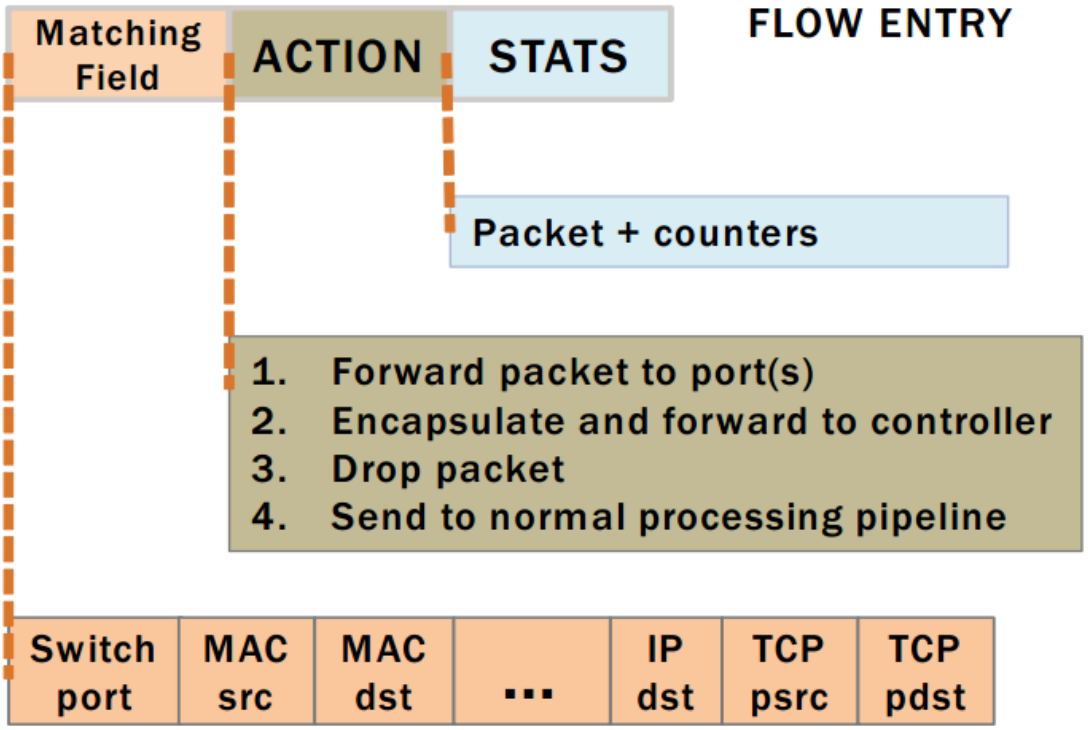
一、SDN 流表查询技术背景介绍

基于网络协议分层的思想，在传统的 TCP/IP 架构网络中，网络转发结点对数据包进行逐层协议处理，各个协议层次在逻辑上相对独立，但同时各层协议的处理方法也变得大相径庭。无论是在软件上还是在硬件上，各层协议的处理逻辑实现都各相迥异。随着网络协议的不断增多，网络协议的层次结构变得越来越复杂，开发新的网络协议、维护各层网络协议之间接口的兼容性等工作都变得越来越困难。因此，寻找一个逻辑上分离，但形式上统一的协议处理架构具有非凡意义。

在软件定义网络（Software Defined Network，以下简称 SDN）中，掌控全局信息的控制器将根据需求制定的流表配置到各个交换机中。在收到一个数据帧后，交换机会查询其匹配的流表项，通过执行该流表项后指定的动作来完成对数据包的处理。一条流表项中包含各层协议头信息，在处理数据帧时，将其与数据帧的各层协议头进行比对从而确定此数据包是否属于该流表项对应的数据流。但在实际情况下，网络中的数据流数量是庞大的（n 台主机间只考虑 IP 域时就能产生 (n(n-1))/2 条数据流），为了节省存储空间，流表项实际上是按照协议域拆分存储的（根据关系数据库的思想，按照协议域拆分存储在减少冗余信息的同时也便于管理）。在流表被拆分为多级流表后，查询一个数据帧所属的流需要进行多次的查询操作，这无疑降低了网络的性能。为解决这一问题，目前使用的最为广泛的方法就是利用 cache 机制进行加速。Cache 中存储着完整的各层网络协议头信息，这样交换

机在收到数据帧后只需要通过一次查询便可以决定该数据帧所属的流和需要执行的操作。由于 cache 的存储空间十分有限，不可能同时存储所有的数据流信息。因此如何维持 cache 的高命中率，保证 cache 与多级流表的一致性，以及防止 cache 抖动等问题便成为了设计 cache 替换策略时需要综合考虑的问题。

1.1 流表的主要结构[1]



首先是匹配域，匹配报文包头中的一个或多个关键字，记录进入交换机的端口，源 MAC 地址，目的 MAC 地址，源 IP 地址，目的 IP 地址，源 TCP 端口号，目的 TCP 端口号。由于 OpenFlow 协议需要匹配网络中所有类型的报文，所以匹配域才会这么的复杂

然后是动作域，对应报文的转发动作。包括转发到制定端口，打

包并发给控制器，丢弃报文，发送给处理管道。

最后是状态域，在这篇文章中，状态域表示某一个表项的状态，常见的状态信息有匹配的次数。

1.1.1 流表存储结构

流表项的存储结构一共有两种，一种叫做 TCAM，一种叫做 SRAM。TCAM 的开销通常都比较大，而 SRAM 利用资源的效率比较高。简单来说，SRAM 实现的存储结构为一个基于 hash 的流表和一个基于通配符匹配的流表的组合。

基于 hash 的流表项，为一个报文的全部的需要匹配的包头域取出一个 hash 值，因此只要某一个域有一点点不一样，就会 miss，就会重新存储一个新的 hash 表项。

基于通配符的流表，如果所有以 W/24 为目的 IP 的报文都要被转发到 1 端口，那么 W 的后 8 位和其他域就不用匹配了 这种方式减小了空间开销，但是在最差情况下，一个报文可能需要遍历整个流表，在表项逐渐增加时，查询时间可能会线性增加。

1.1.2 混合流表结构

基于以上对 SRAM 存储结构中基于 hash 的流表和基于通配符的流表的分析，提出了混合使用两种流表的存储结构，即同时使用基于 hash 的流表和基于通配符的表，首先查 hash 表，没查到再查通配符表，然后将这一个包信息添加到 hash 表中。

这样做的好处是，因为有了基于通配符的缓存表，就尽量减少了返回用户态查询多级流表的次数；因为有了基于 hash 的缓存表，就将查询基于通配符的缓存表的时间复杂度降为 O(1)了。在没有 hash 流表的时候，可能需要遍历整个通配符流表才能找到匹配

但是混合流表结构还是存在问题：首先，流信息变多之后性能会下降，这里是还是说的通配符表的 O(n)查询复杂度问题；其次，需要将通配符表中的信息写入 hash 表，这个写入过程会耗时。同时，hash 表还是会很大，很耗空间。这里说的是 hash 流表中存储的仍然是一个流的全部信息，就像仅仅使用 hash 流表一样耗费空间。

类似于上述混合流表结构，内核态存在两级流表缓存，一级是基于 hash 的流表，一级是基于通配符的流表，报文首先在 hash 流表中进行匹配，如果匹配到了，就根据索引以 O(1)复杂度在通配符流表中取出转发动作；如果没有匹配到，那么就以 O(n)复杂度遍历通配符流表找到转发动作，然后将索引写回 hash 流表。

混合流表与 hash 流表结构的有如下区别：

前者：【Src port | Dest port | Src IP | 。。。 | Src MAC | Dest MAC】+ 【Action】

后者：【hash value】 + 【Action】

hash 流表中值存储匹配域的 hash 值，而不是存储具体的匹配信息。hash 索引通配符流表存储方案的执行顺序是，报文首先在 hash 流表中查询，如果在 hash 表中查到了，还是要继续匹配通配符表项，不能直接取出通配符表的转发动作。如果在 hash 表中取出的动作与

通配符表中取出的动作不一致的话，那么遍历整个通配符表，找到正确的转发动作，并更新 hash 表。

这里隐含两件事情，其一，hash 流表和通配符流表可能存在不一致，再者，通配符表一定是正确的，而 hash 表不一定正确。

二、SDN 中的缓存技术

2.1 FlowShadow[2]

1）记录规则对应的流表项

控制器在向交换机配置规则时，为每条规则制定一个唯一的标识符。在交换机中维护着一张规则和流表项的对应表，记录着哪些流表项中包含着该规则。当网络换发生改变，控制器根据新的网络状况更新了该规则对应的动作时，此交换机可以根据这张表格查找到 cache 中的哪些流表项包含了此项规则，然后将这些流表项全部删除，从而保证了 cache 与多级流表的一致性。

2）记录流表项对应的规则

维护一张规则的状态表，当网络环境发生改变时，控制器在该状态表中给失效的规则打上无效标记。此外，在生成 cache 流表项时，记录下该流表项都包含了哪些规则。对于每个到达的数据帧，交换机在命中 cache 后都将遍历此流表项对应的所有规则的状态，如果发现只要有一项规则有无效标记，便删除此流表项，将数据帧递交给多级流表查询进程。

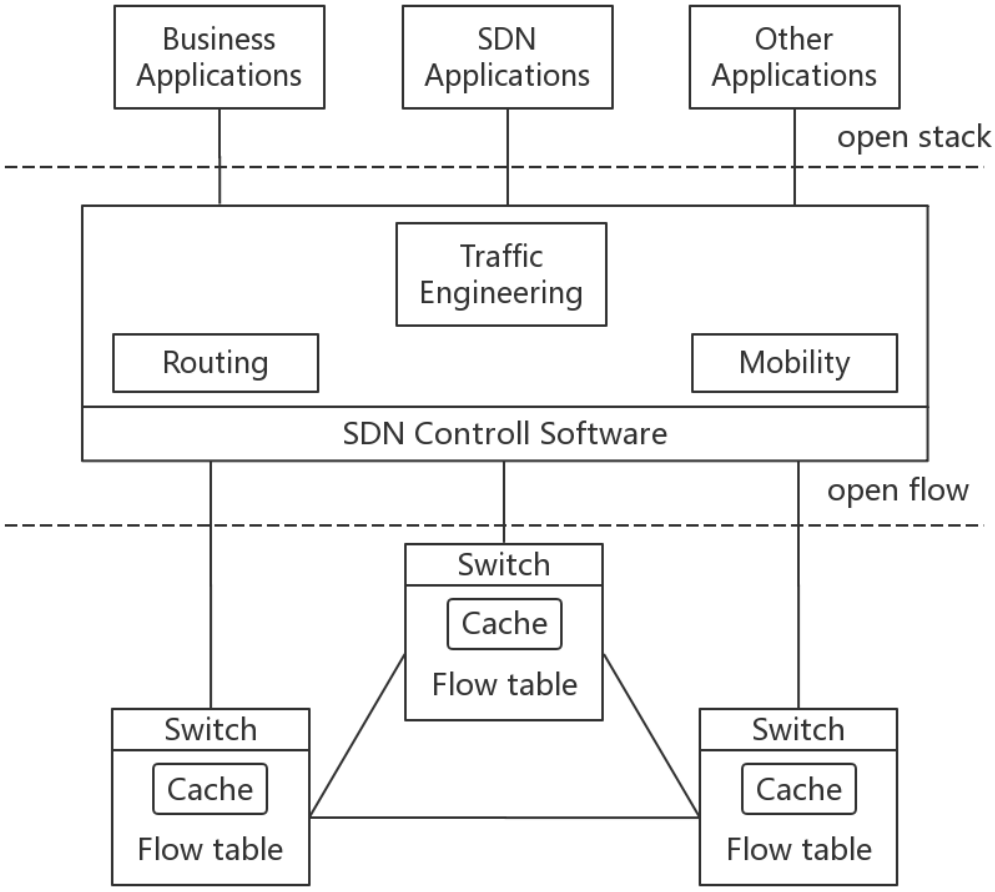
3）记录规则对应的最终动作集

在生成最终动作集时，计算每个最终动作集的哈希值作为该动作集的唯一标识，并记录下每项规则都对应了哪些最终动作集，当网络环境发生改变时，控制器根据该表格给失效规则的相关动作集打上无效标签。为了处理最终动作集哈希值冲突的问题，可以在动作集中维护一个时间属性，记录下该动作集的最近更新时间，在处理数据帧时，如果其匹配的流表项对应的动作集的更新时间晚于此 cache 流表项的生成时间，则可认为此 cache 流表项已经失效。

2.2 Flowmod[3]

此前介绍的技术的关注点都是在 SDN 交换机内部，内核空间对用户空间中多级流表的缓冲问题上。这里介绍一种在 SDN 控制器和交换机间插入缓存模块的技术：通过缓存 SDN 控制器下发的规则更新信息，当到达的数据帧匹配流表项失败时，交换机不再立即向控制器发送请求信息，而是先尝试查询该缓存以获取新的流表项，当该操作失败后才向控制器发送请求信息。

实际上该缓存模块在实现时位于 SDN 交换机内部，可以是硬件缓存也可以是虚拟缓存。SDN 控制器定期地将有更新的规则条目同步到该缓存中。图 1 描述了这种缓存模块的部署方式。

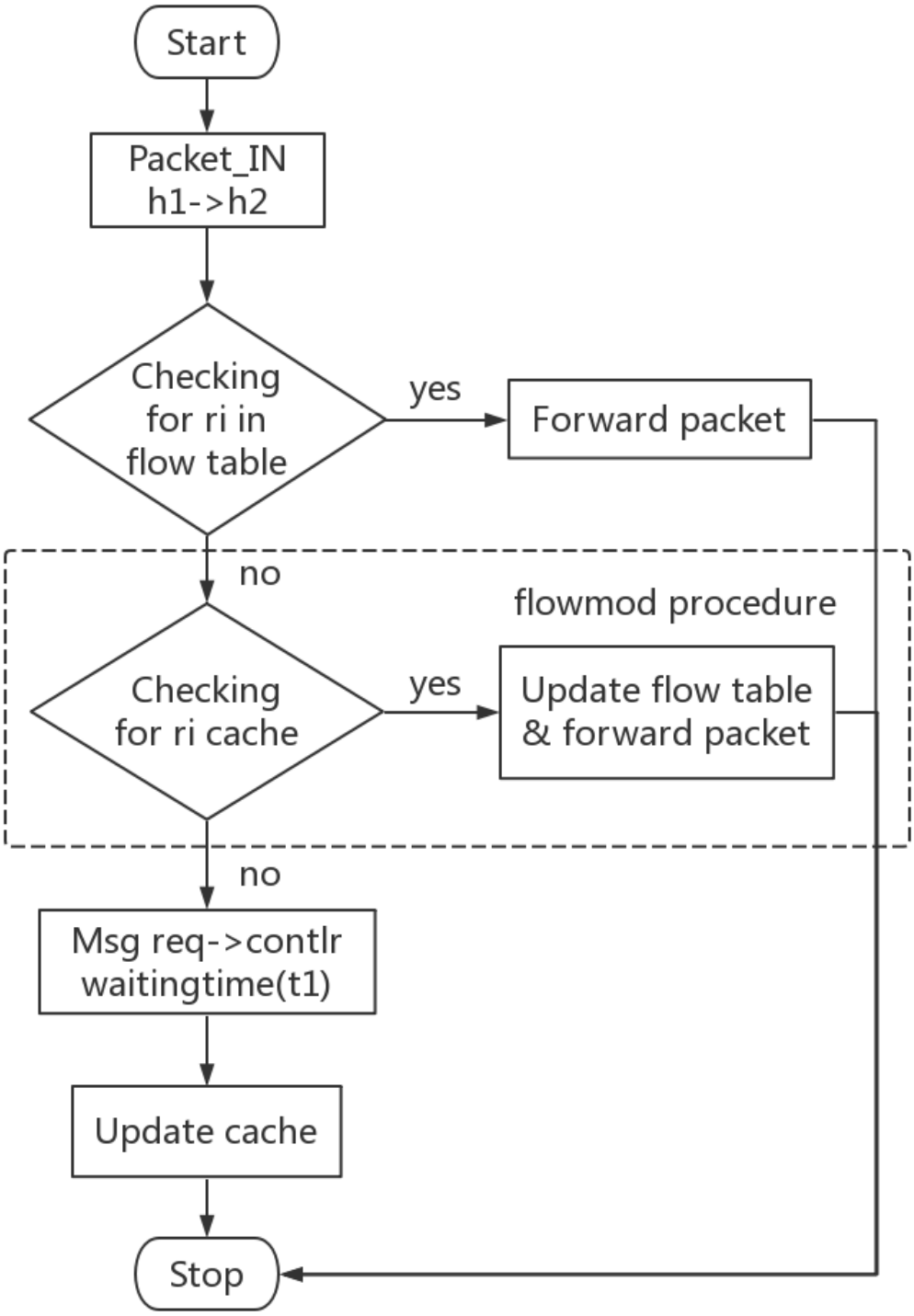


1、将新加入的规则插入到缓存中。

2、将有更新的规则同步更新到缓存中。

3、将失效的规则在缓存中删除。

到达的数据帧在交换机中具体的处理流程如下:



当数据帧在交换机内匹配流表项失败后，首先进入 flowmod 处理流程，只有在从缓存中获取新的流表项失败时才向控制器发送请求。这一机制的引入使得交换机向控制器发送请求的次数明显降低，提高

了交换机的性能，尤其是当数据帧匹配流表项失败时，能显著地降低新流表项建立的延迟。此外还减轻了控制器的处理负载。提升了整个 SDN 的性能。

2.3 Flow-driven rule caching[4]

在找到新的规则替换 TCAM 中缓存的规则时经常会采用 FIFO 或 LRU 算法。但这些方法都有一系列的问题。对于 FIFO 算法，强调先进先出，有些很少用的规则反而在 TCAM 中停留了太长的时间，对于 LRU 算法也只是考虑了 packet 而没有考虑 flow。

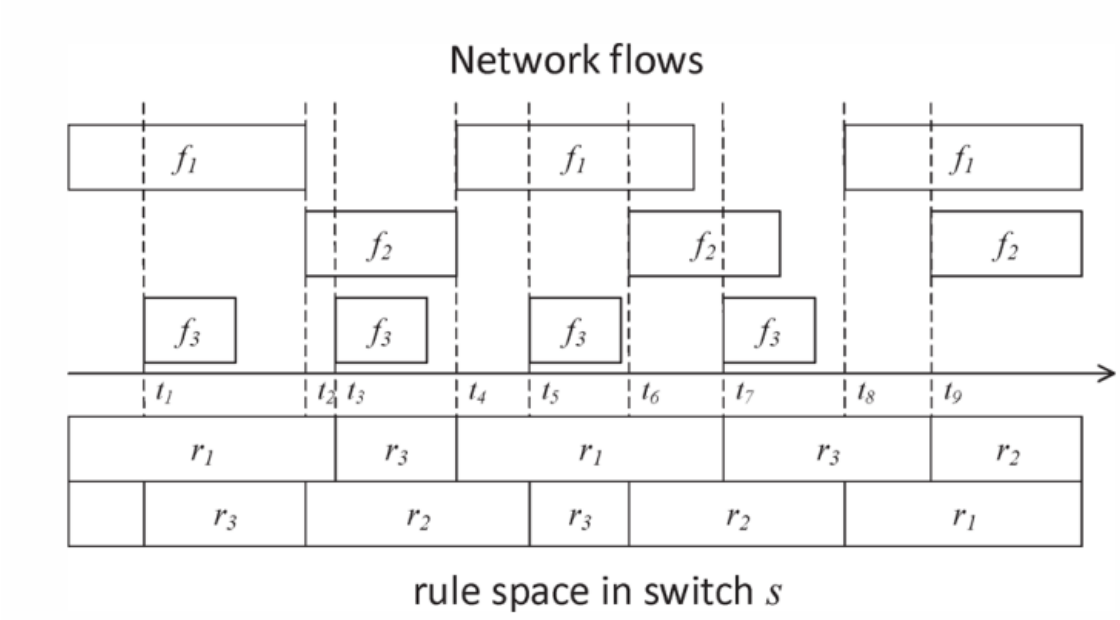


Figure 1: Caching rules for network flows

例如在上图中，TCAM 使用 LRU 来更新 cache，由于不知道每个包的后续情况造成 cache 频繁抖动，整个过程的命中率为 0。

基本思想:

如果缓存一条流的规则，那么整条路线上都要缓存这条规则。每

个流会计算一个计时器 T 预测下次缓存命中的时间，缓存替换时把预测时间最长的规则替换出去

第一步：缓存一条流的规则的时候这条流涉及线路的所有路由器都要把规则添加到 cache 中。

Algorithm 1 Flow-driven caching algorithm

Require: In time t, a packet of flow f_i comes to the switch.

1: **for** $s_j \in S_i$ **do** ▷ Traverse switches in the forwarding path
2: **if** r_i is not (i.e., $X_{ij} = 0$) cached in s_j **then**
3: **if** $\sum_{i=1}^{n} X_{ij} = B_j$ **then** ▷ Cache replacement
4: $m = \arg\max_{k \quad X_{kj}=1} T_k$;
5: remove r_m from cache;
6: **end if**
7: put r_i in cache;
8: Set timer T_i using Algorithm 2
9: **end if**
10: **end for**

第二步：给每一条规则涉及一个计时器 T，如果 cache 满了就把 T 最大的规则替换出去。T 代表这条流的下一个包的预期到达时间。对于可以预测的流，可以把 T 直接设置为下一个包的预期时间，对于未知的不可预期的流，则需要使用算法预测 T，预测的算法表述为：

Algorithm 2 Setting timer T_i for unpredicted flow f_i

```
1: Start timer T_i with an initialized value T_max;
2: while 1 do
3:     if a new packet of flow f_i comes before T_i expires then
4:         set ΔT as the latest packet arriving interval;
5:         start T_i with value ΔT;
6:     end if
7:     if T_i expires then
8:         if ΔT = T_max then
9:             freeze T_i with value T_max;
10:            break;
11:        end if
12:        start T_i with value min(2ΔT, T_max);
13:    end if
14: end while
```

这个算法的意思就是先设置一个最大值，然后计算下一个包到达的间隔设置为 T，到了 T 时间如果没能 hit 就把 T 设置为 2T（但不超过最大值）。

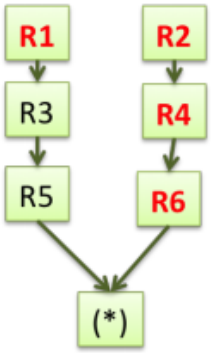
2.4 Infinite CacheFlow[5]

本文的背景是在硬件交换机上使用 TCAM 作为 cache 处理尽可能多的包，如果 cache 没有命中就把包发送给软件交换机，这是因为硬件查询更快，而软件能存储更多的流表规则。然而在 cache 中存储规则时经常会遇到依赖的问题。

Rule	Match	Action	Weight
R1	0000	Fwd 1	5
R2	11**	Fwd 2	5
R3	000*	Fwd 3	20
R4	1*1*	Fwd 4	20
R5	0**0	Fwd 5	90
R6	10*1	Fwd 6	120

Figure 1: Example Rule Table

Rule	Cost
R1	1
R2	1
R3	2
R4	2
R5	3
R6	3



(a) Dependent-set Algo.

一般情况下，我们总希望把最经常使用的规则存放在 cache 中，上图的 weight 就代表每条规则对应的流量。如果只将 R5，R6 存放在 cache 中，那 0000 就会在 cache 中命中 R5，可实际上 0000 应该对应的是 R1，因为 R1 的优先级更高。为了避免这种错误，现有的方案都是把整个依赖关系全部存入缓存，根据优先级来判断命中哪一个。可是许多服务可能会生成很长的依赖链，其中大部分都是很少出现的，但由于优先级更高只能一并被存入 cache。例如防火墙服务，可能在设置允许通过的规则之上还定义了很多禁止的 IP，这些禁止规则很少会遇到，而为了保持 cache 的正确性不得不放入 cache 中，占用了大量空间，影响其他规则放入。在 cache 内和 cache 外交换整个依赖关系链是低效的。

对应上述问题，文章提出的解决方案主要是：

1. 首先生成规则依赖关系的关系图。
2. 把较长的规则链切割为小的规则集，创建一些新的规则来覆盖不常用但优先级更高的规则，如果命中了这个新的规则重新发

给上层软件交换机来处理，只把常用规则和新创建的规则存储在 cache 中，减少了需要存储的规则的数量。

3. 处理规则链的变化

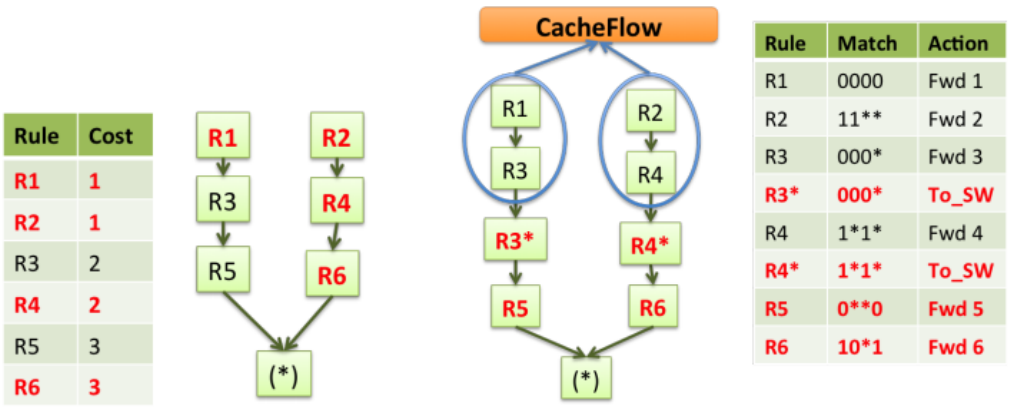
1. 生成依赖关系的关系图

```
// Add dependency edges
procedure add_dependency(P:Policy) {
  deps = ∅;

  // p.o : priority order
  for each R in P in descending p.o
    reaches = R.match;
    for each Ri in P with Ri.p.o < R.p.o
      in descending p.o:
        if (reaches ∩ Ri.match) != ∅ then
          deps = deps  ∪ {(R,Ri)};
          reaches = reaches - Ri.match;

  return deps;
}
```

Figure 2: Building the Dependency Graph

需要注意的地方是它没有找所有的规则之间的关系，而是根据优先级生成规则图，在找到依赖保存关系后就移除当前的 reach，防止更低优先级的规则也建立依赖。

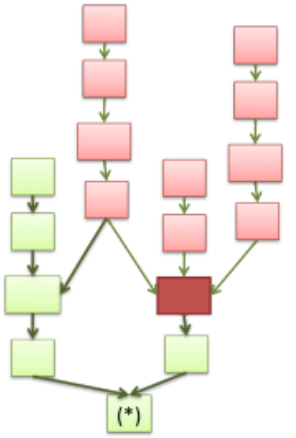
2. 重新生成依赖规则集

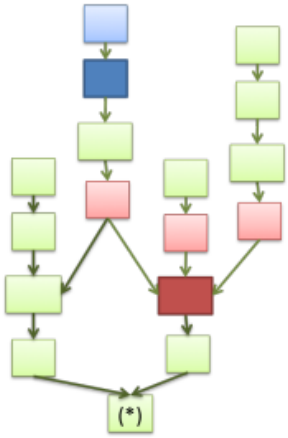


(a) Dependent-set Algo.　　(b) Cover-set Algo.

在生成规则依赖图(a)之后，每一条规则对应一个 cost，cost 就是指这条规则本身和依赖的其他规则数量，计算最优解是 NP 问题，所以构造启发式算法，可以使用贪心算法选出 weight（流量）/cost（数量）最大的规则集放入 cache 中。不过为了解决长依赖链的问题，本文把 weight 较小优先级又较高的规则合并为一个规则 R*，如图(b)所示，R*定义的操作就是转发给上层软件交换机 To_SW，相当于是 miss，这样减少了依赖链的长度，重新计算 weight 和 cost，使用贪心算法选取合并后的规则集放入 cache 中。

3. 处理规则链的变化





(c) Dependent-set Cost (d) Cover-set Cost

规则集之间是独立的，移除旧的规则集不会对新的规则集造成影响。如图（d），移除红色的规则集并不会对新加的蓝色规则集产生影响，作者团队设计了一个算法来维持依赖图的更新，有更新发生时不需要重新构建整个依赖图，但在文章里没有说明具体的算法。

这种方法缩减了规则集，在遇到小流时不会将整个规则链放入 cache 中，而是选取缩减规则链选取 weight/cost 最大的规则集放入 cache 中，避免了 cache 的大规模更新，在遇到大量小流时也能避免抖动，提高了 cache 命中率。

2.5 Elephant flow & Mice flow[6]

2.5.1 大流（elephant-flows）和小流(mice-flows)的定义

数据中心托管各种应用程序，例如 Web 服务，电子商务和社交网络，这些应用程序生成大量的数据中心内部流。关于数据中心交通的最近的研究报告中，将负载量少于 100 Mbytes 称为小鼠流，将负

载量在 100 Mbytes 和 1GBytes 之间的流称为大象流。大多数小流都是由应用程序保持活动数据包（ICMP）或 TCP 确认引起的。此外，MSSQL, HTTP 和 SMB 等应用程序的流量特性与小流而非大流密切相关。

2.5.2 大流（elephant-flows）和小流(mice-flows)的问题

OpenFlow 交换机的一个主要问题是流表的大小有限，在流表容量满时，流表中的一部分流被驱逐。从数据中心流量特征来看，我们观察到大象流非常大（数据量），但与小鼠流相比，数量很少。因此，由于交换机流量表的有限大小导致到控制器的额外流量，因此大象流更可能被驱逐。而大象流往往更加重要。

2.5.3 差分流缓存框架（differential flow cache framework）

经过资料查阅，我们发现了一种差分流缓存框架(differential flow cache framework)，通过快速查找和降低缓存缺失率实现公平和高效的缓存维护。该框架使用基于哈希的放置（hash-based placement）和基于最近最少使用（LRU）的替换机制。

在一条流在流表中 miss 时，它会被返回给控制器进行判断，而产生“控制器的额外数据包输入事件”。控制器的 round-trip 会导致额外的延迟，从而在控制器上产生额外的负载。数据中心中大流会被小流驱逐，从而降低整个网络的性能。

该框架在交换机和控制器之间提出了一个流缓存层（flow cache layer）。对于可扩展的解决方案，该缓存可以与存储多个交换机记录。当一个流在交换机中 match-miss 时，交换机首先查询流缓存，而不是直接联系控制器。

该框架提出了一种高速缓存架构。其中，流缓存层被组织成具有相关索引的动态增长和缩减的块（或桶 buckets）。索引的大小不是固定的，而是取决于当前的桶数。索引值是通过计算该流的相关字段的哈希值获得的（dynamic-index hashing）。

为了使大象流能被更公平的处理，该框架提出了差异化索引机制。该机制通过关联大象流和小鼠流的不同索引到多个 buckets 中，（大象流和小鼠流的索引存在不同的 buckets 中）。在这里要将 buckets 的缓存条目尽可能减少（使用较少的缓存条目进行搜索操作有助于减少使用 SRAM 实现的查找时间）。缓存条目替换策略是 LRU 策略。

差异化缓存框架具有以下优点和功能：

1) 缓存架构和散列函数很简单，因此易于实现。
2) 流缓存的动态增长/收缩（dynamically growing/shrinking）可以让不同的流缓存更好地共享缓存。
3) 差异化索引方法提高了大象流量的公平性，减少了他们的驱逐数量。
4) 缓存的放置，查找和替换机制可确保快速流程处理和高命中率。

1) 流缓存架构（flow cache architecture）

流缓存层由多个交换机使用，因此流缓存层由许多交换机的流缓存组成。交换机仅映射到一个流缓存，这避免了大型网络中高速缓存模块之间协调需求。流缓存层被组织为一组固定大小的 buckets。一开始时，buckets 的数量被定义为预定值， 随后，根据缓存条目的数量动态的调整 bucket 的数量。

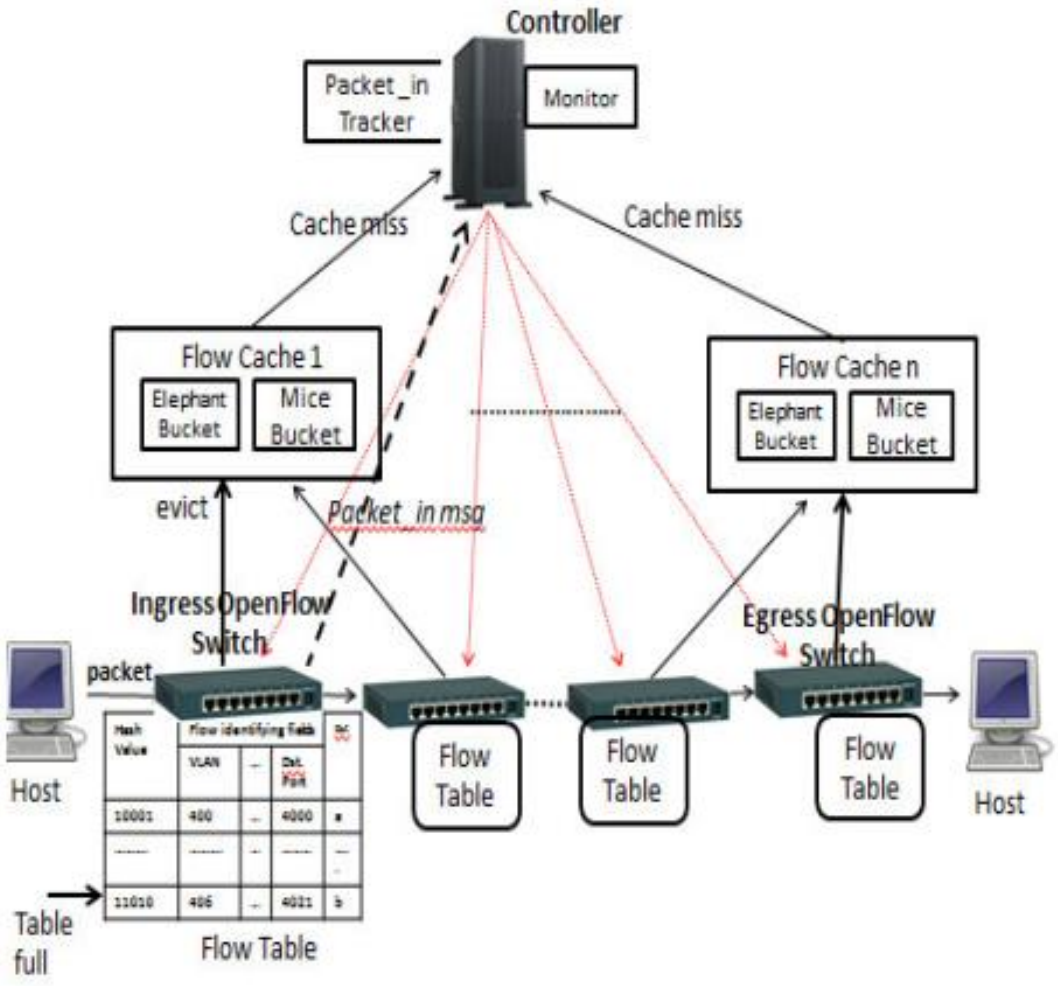


Figure 1. Interactions between the controller, cache and flow table

图 1 显示了 OpenFlow 交换机，流缓存和控制器之间的交互。 由于容量溢出或基于超时机制，将逐出流表中的条目。这些条目将存储

在关联的流缓存中。当数据包到达启用 OpenFlow 的交换机时，首先在 OpenFlow 交换机的流表上执行查找，流表未命中将查找流缓存；如果流缓存中存在匹配，则执行相关的数据包转发操作；否则，packet_in 事件被发送到控制器并在流表中插入一个条目。如果缓存桶已满且最大桶数已经用完，则新的流条目会导致缓存未命中导致现有流的逐出。

2) 桶的动态索引哈希（Dynamic-index hashing）

流缓存被组织成用于小鼠流和大象流的单独桶。虽然流缓存中的桶的最大数量受到约束，但是大象流和小鼠流使用的桶的数量是根据需求动态确定的。我们使用动态索引哈希将流映射到存储桶。

与流相关联的 k-bit 索引是动态的，并且 k 的值取决于当前用于该类型流（小鼠或大象）的桶的数量。 与每个桶相关联的是一个索引，它存储具有相同索引的流（鼠标或大象，但不是两者，大象流和小鼠分开放置）。 我们使用流行的 SHA 散列方法，从随机散列值中提取 x 位（用于形成索引）更有可能保持桶的平衡。

我们使用一个目录来存储索引值和指向相应存储桶的指针。 索引目录采用数组的形式，索引最多有 2b 个条目，每个条目存储一个存储区地址。 变量“b”被称为目录的最大深度，对应于最大的桶数。图 4 是大象流和小鼠流分别只有一个桶的情况（BucketA，索引为'0'，用于存储小鼠流；BucketB 索引为'1'，用于存储大象流）。

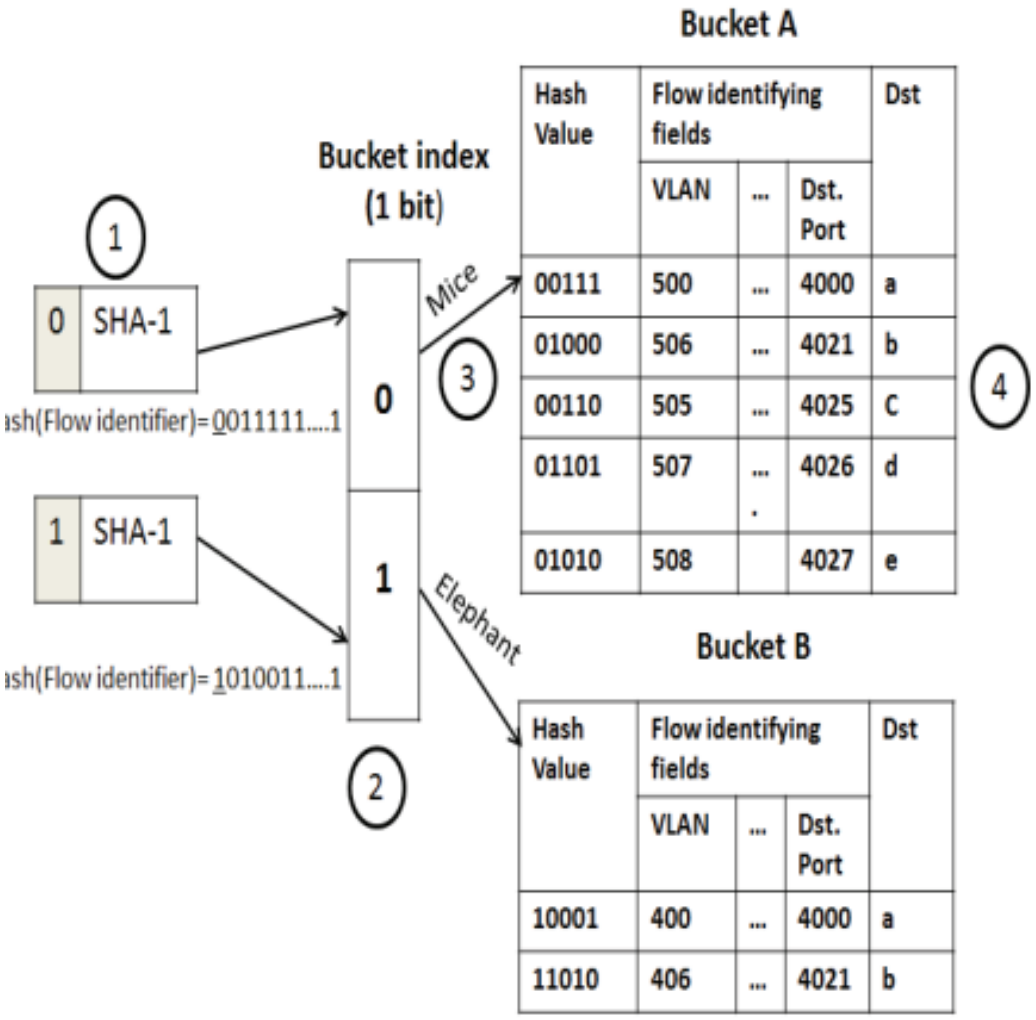


Figure 4. Illustration of buckets and Index tables

当一个桶的数量已经满了，而有一个新的条目要加入时，桶扩张机制（bucket expansion mechanism）将创建一个新桶。

3) 桶扩张机制（Overflow handling- Bucket Expansion）

如果存储桶溢出，则使用一个额外比特位新索引将存储桶大小加倍。例如，如果最初使用 1 位索引，则将使用扩展 2 位索引。因此，原始存储桶中的条目（例如，索引为 0）将分布在两个存储桶（索引

00 和索引 01）中。索引目录也将加倍，即，如果它最初包含 2k 索引值，它现在将包含 2k + 1 个条目，这意味着深度增加到 k + 1。我们注意到这些 k + 1 比特是由随机哈希值形成的。因此，新铲斗可能更加平衡。如果 0 索引的小鼠桶超过其容量（例如 1024），它将扩展为 2 位索引桶：'00'和'01'，从而将其最大容量增加到 2048。如果需要进一步扩展，它将扩展到 3 比特索引桶：'000'，'001'，'010'和'011'。类似的程序用于前缀为“1”的大象桶。我们注意到鼠标流和大象流的索引位数不必相同。

桶扩展示例如图 5 所示。这里我们假设桶大小为 5。当第六个小鼠流到达时，桶 A 溢出并使用新的桶 A'，并且六个条目（包括新的条目）分布存储在桶 A 和 A'，每个都使用 2 位索引。

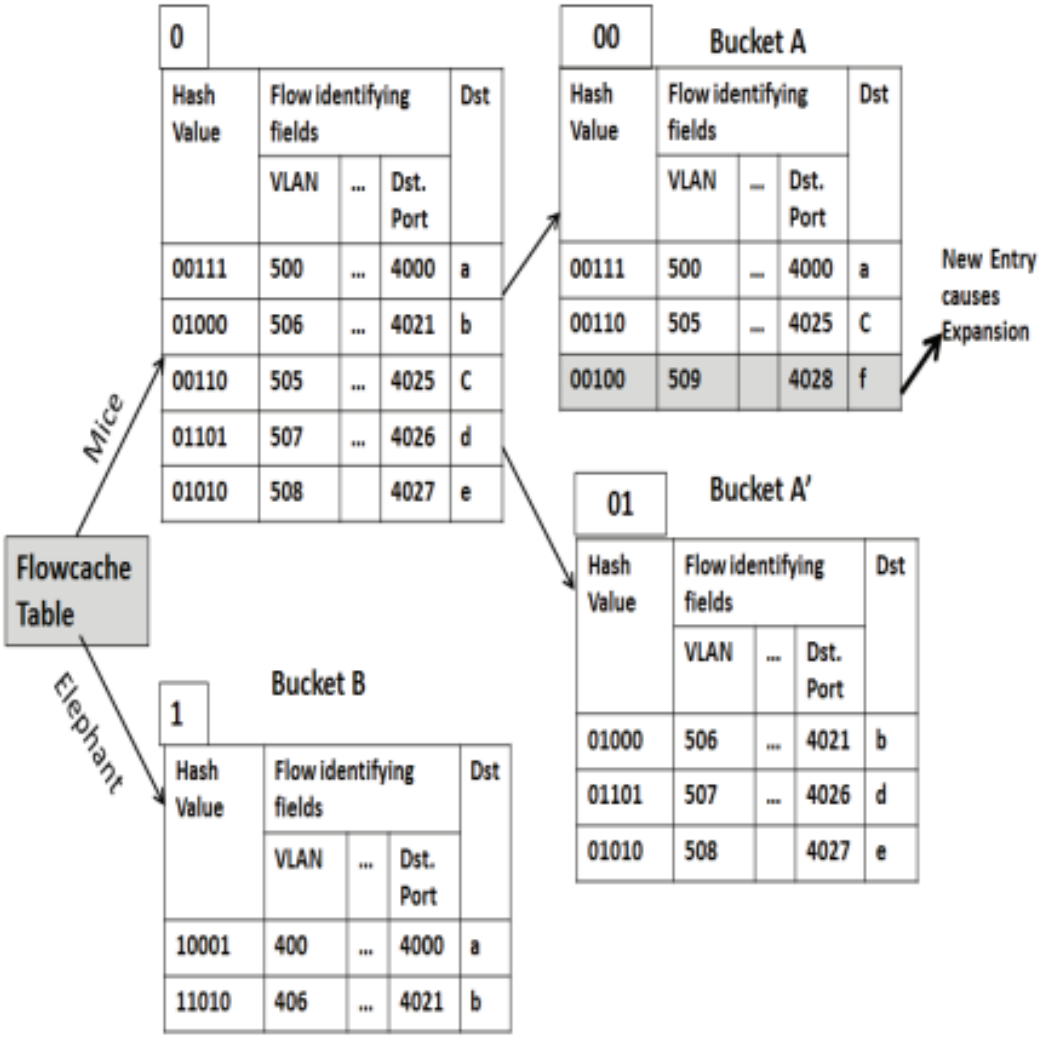


Figure 5. Illustration of Bucket Expansion

2.6 Dependency-Aware Rule-Caching[7]

2.6.1 TCAM 和软件交换机

软件定义网络（SDN）允许控制应用程序在底层交换机中安装细粒度转发策略。虽然三元内容可寻址存储器（TCAM）可以在具有灵活通配符规则模式的硬件交换机中实现快速查找，但成本和功耗要求限制了交换机可以支持的规则数量。而且，TCAM 硬件交换机无法维

持规则表的高速率更新。

软件交换机虽然在更新流表规则上有优势，但是其匹配速率和吞吐量等都远不及硬件交换机。

通过查阅文献，我们发现了一种通过结合最佳的硬件和软件处理，为应用程序提供高速转发，大规则表和快速更新的方法，称为CacheFlow 系统。它“缓存”小型 TCAM 中最常用的规则（rule），同时依靠软件交换机来处理少量的“缓存未命中”流量。由于 TCAM 的规则集存在着依赖链。CacheFlow 不是缓存大型依赖规则链，而是“拼接”长依赖链以缓存较小的规则组，同时保留策略的语义。

该CacheFlow原型实验证明规则拼接可以有效利用有限的TCAM空间，同时可以快速适应策略和流量需求的变化。

2.6.2 Cacheflow 缓存系统的架构

我们的 CacheFlow 架构由 TCAM 和分片软件交换机组成，如图 1 所示。软件交换机可以在数据平面上的 CPU 上运行。

CacheFlow 由 CacheMaster 模块组成，该模块从 SDN 控制器接收 OpenFlow 命令。CacheMaster 保留 OpenFlow 接口的语义，包括更新规则，查询计数器等的能力。CacheMaster 使用 OpenFlow 协议将规则分发给未修改的商品硬件和软件交换机。CacheMaster 是一个纯粹的控制平面组件，控制会话显示为虚线，数据平面转发显示为实线。CacheFlow 通过将规则集分成两组来实现这些高速转发——一组驻留在 TCAM 中，另一组驻留在软件交换机中。

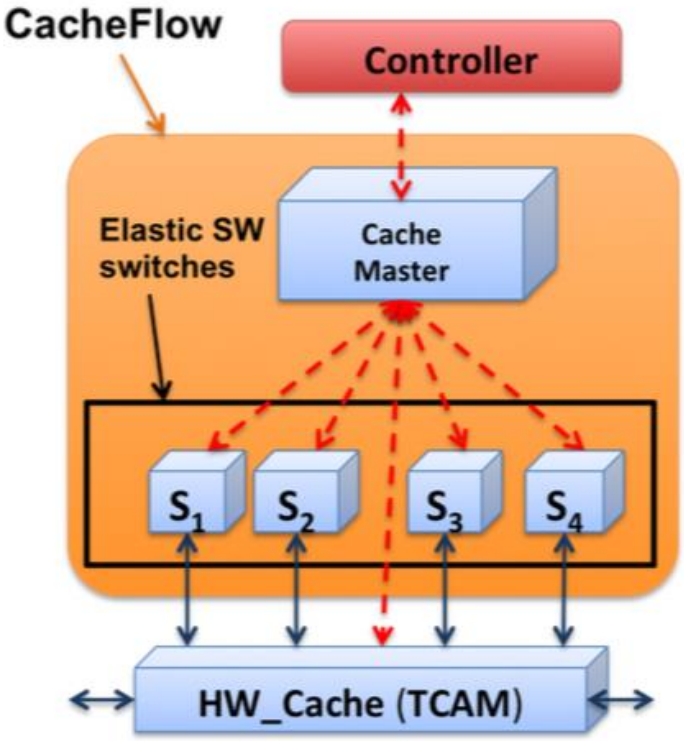


Figure 1: CacheFlow architecture

为了处理规则依赖性，我们构建了一个给定的优先级规则列表的表示，作为一个循环的有向无环图（DAG），并设计了增量算法，用于为该数据结构添加和删除规则。我们的缓存替换算法使用 DAG 来决定在 TCAM 中放置哪些规则。 为了保留与大部分流量匹配的规则表空间，我们设计了一种新的“splice”技术，将长依赖链分割成等语义的几个短依赖链。splice 创建了一些“覆盖”大量不常用的规则的新规则，以避免对缓存进行监控。该技术扩展到处理规则的变化，以及随着时间的推移它们的受欢迎程度的变化。

2.6.3 规则依赖的构建与更新

Rule	(dst_ip, dst_port)	Ternary Match	Action	Priority	Weight
R1	(10.10.10.10/32, 10)	000	Fwd 1	6	10
R2	(10.10.10.10/32, *)	00*	Fwd 2	5	60
R3	(10.10.0.0/16, *)	0**	Fwd 3	4	30
R4	(11.11.11.11/32, *)	11*	Fwd 4	3	5
R5	(11.11.0.0/16, 10)	1*0	Fwd 5	2	10
R6	(11.11.10.10/32, *)	10*	Fwd 6	1	120

(a) Example rule table

(b) Incremental DAG insert (c) Incremental DAG delete

捕获规则表中所有依赖关系的简明方法是构造一个有向图，其中每个规则都是一个节点，每个边捕获一对规则之间的直接依赖关系，如图 2（b）所示。 在以下条件下，子规则 Ri 和父规则 Rj 之间存在直接依赖关系：如果从规则表中删除 Ri，则应该命中 Ri 的数据包现在将命中规则 Rj。

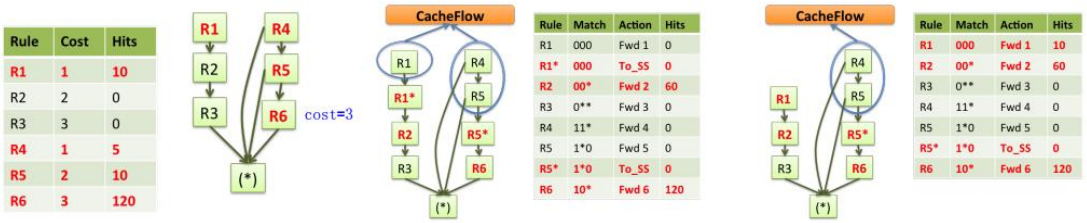
为了确定规则与规则之间的关系，对于任何给定的子规则 R，我们需要找出匹配 R 的数据包可以达到的所有父规则。这可以通过获取与 R 匹配的符号集合并通过所有规则迭代它们来实现。为了找到直接依赖于 R 的规则，该算法按照优先级递减的顺序扫描优先级低于 R 的规则 Ri。对于每个扫描的新规则，它确定与该规则关联的谓词是否与可以到达该规则的数据包集相交。如果是，则存在依赖性。

上述算法在更新时有着严重的效率问题，通过观察我们发现上图（b）我们发现，其实只用扫描左半边结点。因此提出了一种增量更新 DAG 图的算法。当更新一个结点时，不去打扰与它没有直接或间接依赖的其他结点。该算法时 DAG 图的更新速率得到了大幅度减少。

2.6.4 缓存机制

为了确定将哪些规则放入 TCAM 中，我们用两个指标来衡量：为每个规则分配一个 cost，对应于必须一起安装的规则数量；和一个

weight“对应于预期达到该规则的数据包数量。我们的缓存机制就是再 TCAM 能容纳的总 cost 一定时，如何使 weight 总量最大。



(a) Dependent Set Algo. (b) Cover Set Algo. (c) Mixed Set Algo

由于规则之间存在依赖，当我们决定要把 r6 存入 TCAM 中时，也需要把其依赖的 r4 和 r5 同时存入 TCAM。在这里有三种缓存机制。

$$\textit{Maximize} \quad \sum_{i=1}^{n} w_i c_i$$

$$\text{subject to} \quad \sum_{i=1}^{n} c_i \leq C;\ c_i \in \{0,1\}$$

$$c_i - c_j \geq 0 \text{ if } R_i.is\_descendant(R_j)$$

1) 贪心原则（Caching Dependent Rules）

在我们的贪心算法中，在每个阶段，算法选择一组规则，最大化组合规则权重与组合规则成本（ΔW/ΔC）的比率，直到总 cost 达到 TCAM 的容量 k.此算法的时间复杂度为 O（nk）。在图 4（a）中的示例规则表中，贪婪算法首先选择 R6（及其依赖集 R4; R5），然后选择 R1，使总 cost 为 4.因此，TCAM 中的规则集为 R1;R4; R5 和 R6 是最佳的。

2) 覆盖原则（Splicing Dependency Chains）

上面的方法虽然运算速度快，但是可能导致一个问题：可能会有大量 low-weight 的低优先级 rule 由于依赖被写入 TCAM。因此，又提出了一种覆盖原则：通过创建少量涵盖许多低权重规则的新规则来拼接“依赖链”，并将受影响的数据包发送到软件交换机。

对于图 4（a）中的示例，我们不是为 R6 选择所有依赖规则，而是计算覆盖命中 R6 的数据包的新规则 额外的规则将这些数据包转发到软件交换机，从而打破了依赖链。例如，我们可以安装匹配“1*1*”的高优先级规则 R5*和操作 forward_to_Soft_switch，将 R5*和 R6 一起写入 TCAM 中。这样，TCAM 的 cost 得到了减少。 同样，我们可以创建一个新的规则 R1 *来打破对 R2 的依赖。这种算法避免安装像 R4 这样的优先级较低，weight 较轻的规则到 TCAM 中。

3) 混合原则（An Optimal Mixture）

上述的方法虽然减少了 cost，但也可能会减少 weight。例如，为了缓存图 4（c）中的规则 R2，依赖集算法是更好的选择，因为 TCAM 中的依赖集处理的流量更高，而成本与 cover 集（R2*）相同。为了综合贪心算法和覆盖算法的优点和缺点，平衡 cost 和 weight，在每次迭代时混合原则选择两种算法中最好的算法。因此，我们考虑选择两组中最好的一个度量，即 max（ΔWdep /ΔCdep;ΔWcover /ΔCcover）。此算法称为混合集算法。

三、ovs 中的流表查询技术[8]

OvS 是一个通用网络应用，需要支持所有类型的需求，例如 L2 MAC 地址的匹配，L3 IP 地址（DIP 和 SIP）的匹配，L4 端口匹配，生存时间限制，包头域修改等；

OvS 强调的是对一般情况的支持，而不是对最坏情况的支持，也就是说，OvS 只需要在平均情况下达到良好性能；

OvS 布置在网络的边缘，随时会进行交换状态（流表）的更新；

OvS 里有两个主要构件用来进行包转发，一个是 ivs-vswitchd，这个各平台通用，另一个是 datapath kernel module，这个平台独立；前者在用户态，后者在内核态，当一个报文到达的时候，首先在内核态中查找匹配项，如果没有找到，就转发到用户态的 ivs-vswitchd，然后将这一条表项添加到内核态中

micorflow cache 是针对一条流建立的缓存表项，megaflow 是针对一系列流的集合建立一个缓存表项，这个集合中的所有流都有相同的转发动作，而且都可以通过一个前缀匹配到；在有了 megaflow cache 之后，microflow cache 存储的实际上不再是转发表项，而是 megaflow cache 的索引；

元组分类搜索：元组指的是报文头部中每一个需要匹配的域，例如 SIP，DIP，MAC 等等，每一个元组都会建立一个哈希表，新来的报文通过匹配每一个元组，得到转发动作。元组分类搜索支持 O(1)时间内的更新操作，支持任意形式的报文的匹配而不需要改变匹配算法

本身，使用 O(1)内存空间

7.OvS 提出了“resubmit”方法，解决了“交叉相乘问题”：如果元组空间搜索要匹配两个域 A 和 B，A 中有 n1 条表项，B 中 n2 条表项，如果要用一个表为所有的流建立缓存，那么缓存中一定会有 n1*n2 条表项。如果 A 和 B 是无关的，那么一定有一些 A 与 B 的叉乘项是永远不会被匹配到的，因此可以采用两个表来分别缓存 A 和 B，然后将 A 和 B 的匹配结果结合起来得到转发动作

microflow 的来历：

OvS 创建之初，因为发现内核态处理报文能够得到非常的快，因此将所有的报文处理全部放到内核态进行，也就是全部列表项都放在内核态中。但是因为在内核态开发十分的麻烦，而且每次更新到涉及到内核的部署和更新，于是这一方法被放弃了。

当前（2.0）版本的 OvS 使用了一种新的方法，用 microflow cache 实现内核模块，缓存中的表项是报文全部的域的精确匹配项。但是使用这种方法有很多问题，其中一个是流的建立非常的慢，因此采用了攒多个流一起建立缓存，和使用多线程利用多核优势的方法

mageflow cache：支持通项匹配；所有表项没有优先级；匹配只在一个表中进行，而不是一系列的表连续匹配。因为 OvS 要支持所有类型报文的匹配，因此 megaflow cache 中必须包含所有需要匹配的包头域，因此，megaflow cache 中可能会有无用的匹配项，无用表项多了之后，就会使得比配速度变慢。为了优化，采用了四种方法来尽量减少元组数量：

1）元组优先排序：先匹配优先级高的字段，匹配到了就可以停止匹配

2) 阶梯查询: megaflow 只匹配包头域的子集, 而不需要匹配 tuple 中的所有字段。确定子集的方法如下:

将所有的域划分为四组：元数据，L2，L3，L4，然后划分四个检查等级： 一、检查元数据 二、检查元数据和 L2 域 三、检查元数据、L2、L3 域 四、检查所有域 四个等级依次检查，只有匹配成功的等级里面的域会被添加到 megaflow table 中，其他域则不用检查

3）前缀查询：使用字典树匹配 IP 字段，避免匹配 IP 地址的全部 32 位

4）分类器分割：为每一个域创建一个典型元数据，这个元数据匹配所有的合格流，如果一个流不符合元数据，那么将不会符合下一步分类的任何结果

缓存有效性检查：因为流表更新等原因，缓存可能会需要更新。

优先级：当有高优先级的路由表项加进来的时候，原先存储在缓存里面的所有低优先级表项都要修改

大范围更新：每次在缓存中命中之后，都返回用户态查流表，比较两次匹配的结果，如果不一样，则更新缓存;

小范围更新：为每一个域的每一项都给定一个标签 tag，将所有改变了的域的项的标签加入一个 tag 集合，周期性的遍历缓存和 tag 集合，如果发现缓存表项中有 tag 集合中对应的域，那么更新这一个表项。因为 OvS 控制器的应用会不断变复杂，而且 OvS 中会增加更多

会随着网络状态改变的转发动作，每一个周期都会更新更多的表项，因此这一方法被 OvS 放弃了。

大范围更新的问题：

在更新缓存表项的时候，不能为新来的流建立表项，这些流所包含的报文都会被阻塞。解决办法是采用多线程，使得更新缓存的时候还能建立新的表项。为了提高大范围更新的速度，采用多线程建立缓存表项。但是流的删除还是单线程，因此可能会出现流建立速度超过流删除速度，导致内核空间被迅速充满。OvS2.1 中对流的删除也采用了多线程，使得流的删除的速度域流的建立速度大致相等。

流的老化：

缓存的流表项不能一直存储在内核中，一些很少用到的缓存表项应该被删除。目前采用的是生存时间，如果一个表项在一定时间内没被使用，那么就删除这个表项。

参考文献

[1] Ha N, Kim N. Efficient Flow Table Management Scheme in SDN-Based Cloud Computing Networks[J]. Journal of Information Processing Systems, 2018, 14(1).

[2] Wang Y, Tai D, Zhang T, et al. FlowShadow: Keeping update consistency in software-based OpenFlow switches[C]//Quality of Service (IWQoS), 2016 IEEE/ACM 24th International Symposium on. IEEE, 2016: 1-10.

[3] Anju K B, Reju J, Namboothiri L V. Enhancement of Flow in SDN by Embedded Cache[J].

[4] Li H, Guo S, Wu C, et al. FDRC: Flow-driven rule caching optimization in software defined networking[C]//2015 IEEE International Conference on Communications (ICC). IEEE, 2015: 5777-5782.

[5] Katta N, Alipourfard O, Rexford J, et al. Infinite cacheflow in software-defined networks[C]//Proceedings of the third workshop on Hot topics in software defined networking. ACM, 2014: 175-180.

[6] Lee B S, Kanagavelu R, Aung K M M. An efficient flow cache algorithm with improved fairness in software-defined data center networks[C]//Cloud Networking (CloudNet), 2013 IEEE 2nd International Conference on. IEEE, 2013: 18-24.

[7] Katta N, Alipourfard O, Rexford J, et al. Cacheflow: Dependency-aware rule-caching for software-defined networks[C]//Proceedings of the Symposium on SDN Research. ACM, 2016: 6.

[8] Pfaff B, Pettit J, Koponen T, et al. The Design and Implementation of Open vSwitch[C]//NSDI. 2015, 15: 117-130.