CoVisor：A Compositional Hypervisor

for Software-Defined Networks

# 一、文章概述

Covisor是一种网络管理程序。在一个网络里，使用不同语言编写的、部署在不同控制平台的控制器管理网络不同的区域，Covisor组合了这些控制器，使得它们能够协同工作，管理整个网络。Covisor还对网络拓扑进行抽象化，每个控制器看到的都是一个虚拟的网络拓扑，网络管理员可以指定每个控制器的权限。文章最主要的贡献是建立了一套高效率的算法来整合不同的控制器、把虚拟的网络拓扑编译为实际的物理网络拓扑，以及高效率地处理规则更新。

# 二、背景介绍

SDN的一个重要原则是解耦，管理员可以部署最适合需求的硬件和软件，而不是受各种兼容性限制，因此需要一个网络管理程序来组合这些孤立的控制器。这样的管理程序需要做到：

1. 集成多种不同的控制器
2. 定义抽象的网络拓扑，每个控制器都可以看到分配给自己的拓扑结构
3. 防止失效控制器对系统的影响，控制器有一定的访问权限控制，不能访问或修改某些字段

现有技术的主要挑战是效率问题，文章定义了一系列算法来解决这些问题，文章的主要贡献包括：

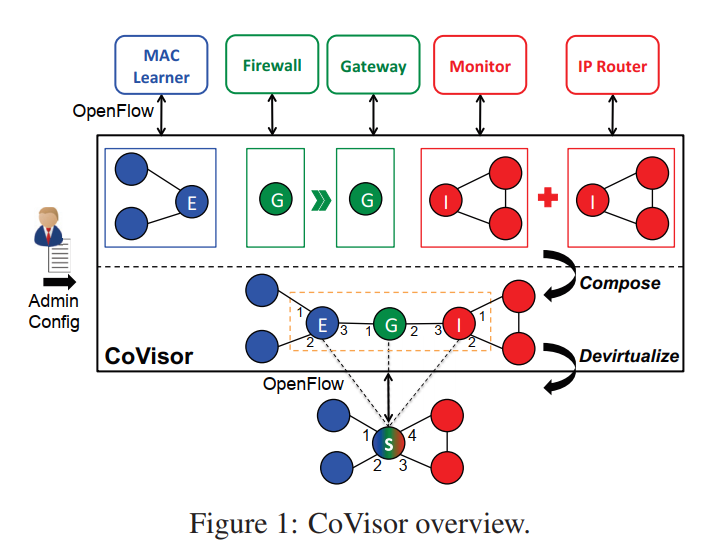
1. 定义了组合多种控制器的管理程序的体系架构
2. 设计新的并行、串行和覆盖算法来组合控制器的操作
3. 设计新的增量算法来把为虚拟拓扑编写的规则编译为实际物理交换机的规则
4. 使用自定义的数据结构来进行访问控制

# 三、Covior简介

为了实现以上特征，Covisor主要分为两个阶段：

1. 整合孤立控制器的策略，变成全局的组合策略
2. 把这个组合策略编译为物理网络能理解的策略

Covior的架构图如下图所示：



Covisor接收管理者的网络配置，这些配置定义了如何集成不同控制器的策略（是先运行A，再运行B，还是A与B共同执行，还是A的规则覆盖B的规则），还定义了每个控制器的虚拟拓扑的创建方式，以及每个控制器的访问权限。

对控制器组合策略的配置包括：

**Actions**：最基本的包组合策略就是原子的包处理操作Action，如丢包操作drop，指定从某个端口转发的操作fwd(3)等等。

**Parallel operator(+)**：并行操作符，相当于把一个包复制两份，同时给控制器T1和T2执行，再将计算的结果进行综合。

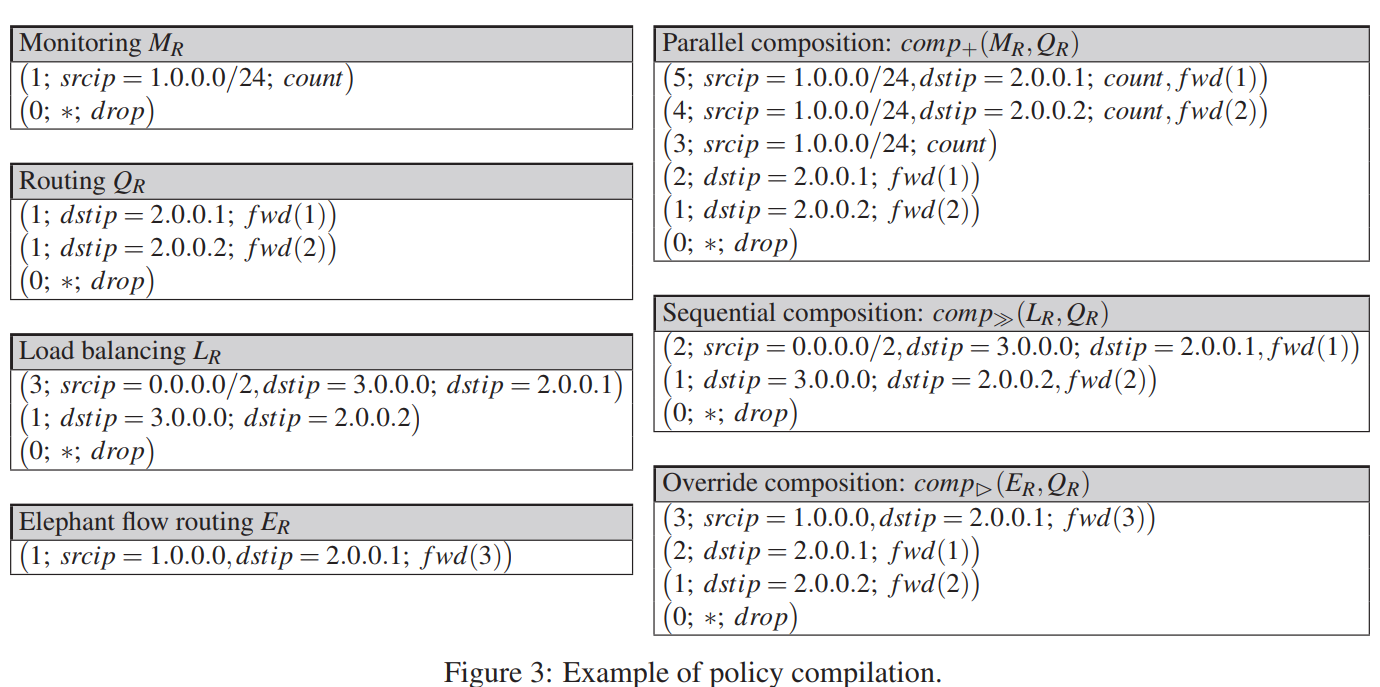
**Sequential operator (>>)**：串行操作符，一个控制器在处理完以后再给下一个控制器进行处理。

**Override operator ()：**优先级更高的规则优先执行

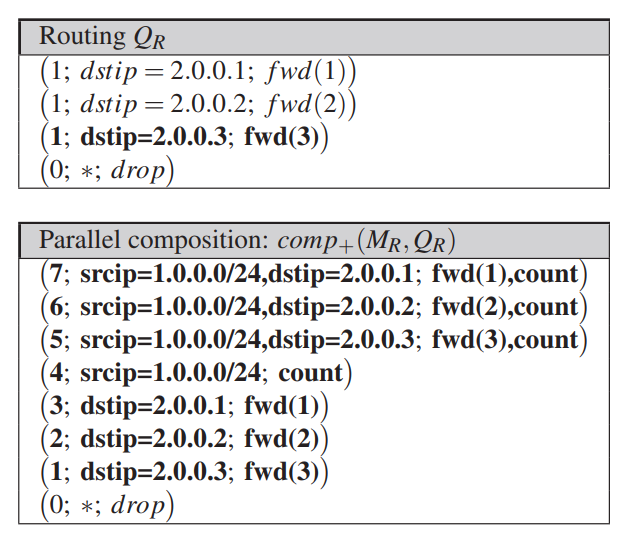
对控制器的限制包括：

1. 每个控制器只能看到虚拟的网络拓扑，不同控制器看到的网络拓扑是不一样的
2. 每个控制器只有权限访问部分包信息

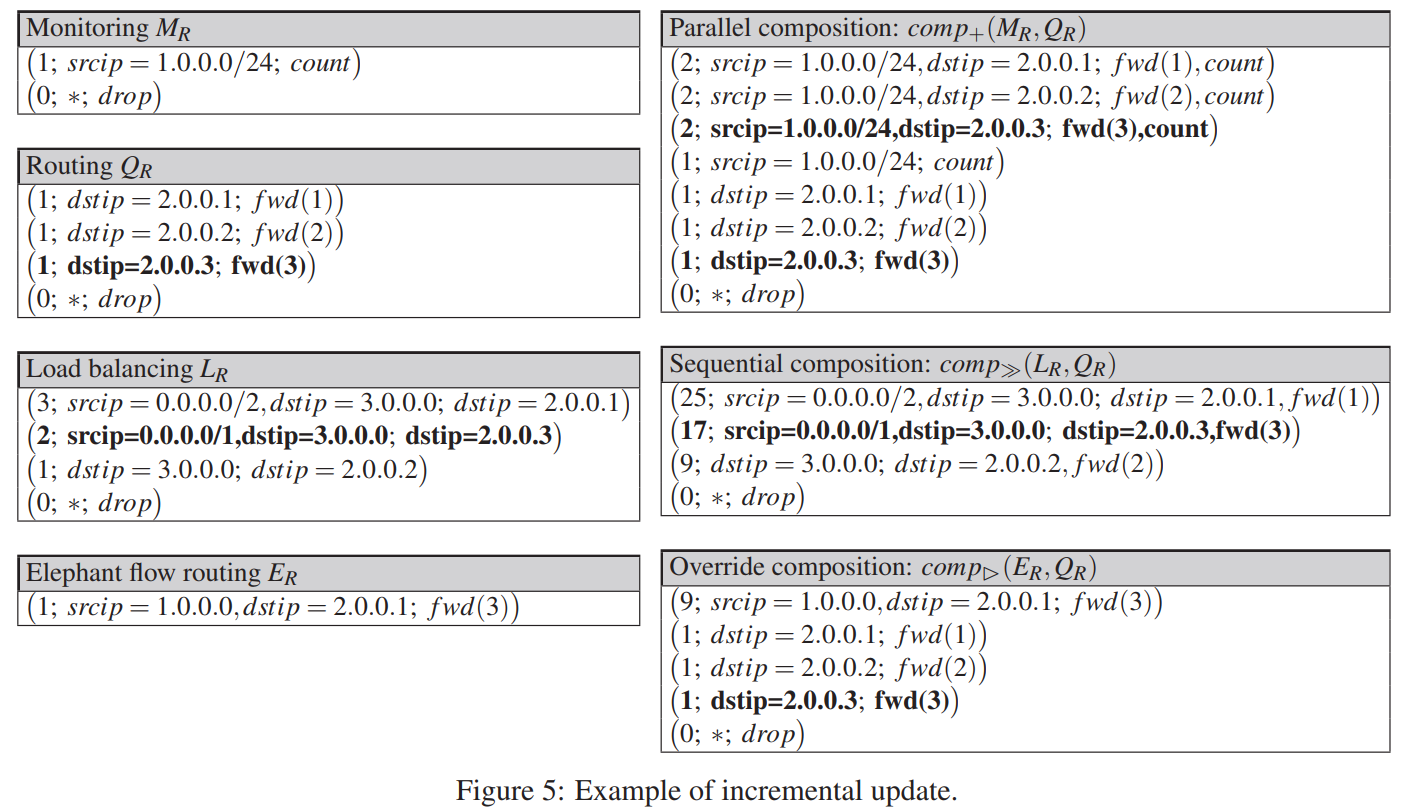
# 三、增量组合策略



对于并行操作，需要两两组合两个控制器规则T1和T2中的所有规则，将这些组合的规则放在更高的优先级上。对于串行操作，需要检查第一个控制器规则T1中的规则，将对应的规则与T2中的规则进行合并。对于覆盖操作，需要将第一个控制器规则T1的所有规则放在更高优先级的位置。

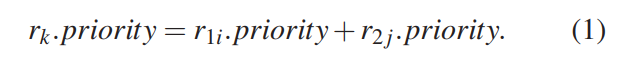


如果只是顺序从0开始到高位依次排列优先级，在规则更新时会造成很大的负担。如下图所示，在QR增加一条新规则时，集成规则集MR+QR中大部分的组合规则的优先级都需要更新，这种方法需要的计算量太大。因此需要一种新的计算优先级的方法。

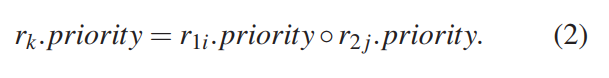


如图所示，文章这样定义组合规则的优先级：

如果是并行操作，那么优先级定义为：



如果是串行操作，那么优先级定义为：

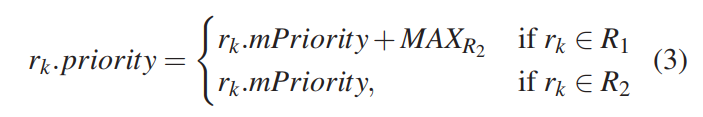


其中小圆圈代表了拼接操作，也就是直接把两个优先级的数字当成字符串拼接在一起。但是这样的拼接是在一定的进制下进行的，在10进制下，1和2拼接为12，在8进制下，1和2拼接为1\*8+2=10。因此优先级的实际定义为：



其中MAXR2指的是第二个规则集T2的最大规则数量

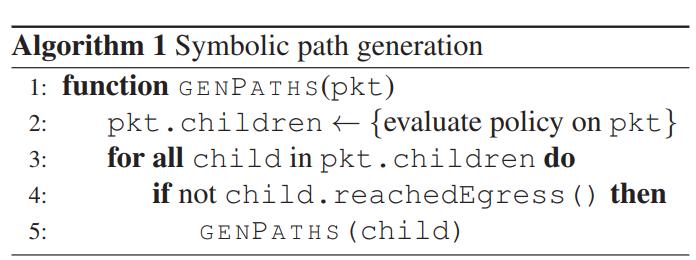
如果是覆盖操作，那么优先级的定义为



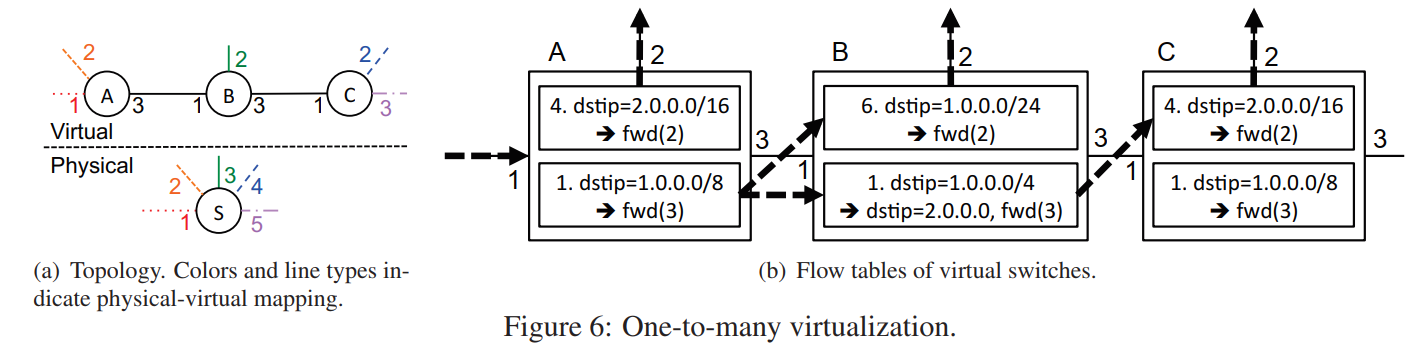
使用这种方法定义优先级，在更新到来是只需要更改几个响应的规则，其他规则的内容和优先级都不会有影响.

# 四、编译拓扑转换

文章重点讨论了一对多的转换，也就是一个物理交换机对应多个虚拟交换机的转换。算法需要从多组虚拟规则中编译出实际物理设备需要的表项。



首先从一个入口开始发送符号包如果这个包没有到达出口，则递归寻找子结点。



例如，对于A、B、C三个规则集，首先发送两个符号包：P1：2.0.0.0/16 ，P2：1.0.0.0/8，其中P2没有到达出口，所以生成P21和P22， P21在B中匹配第一个规则，从2离开，P22在C中匹配第一个规则，从3离开。

由此生成三条路径：

P1： A1

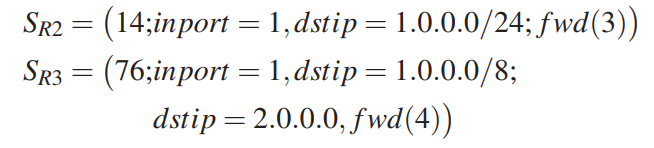
P21 ：A2 → B1

P22 ：A2 → B2 → C1

通过这些路径来进行规则集的合并，如A2 → B1合并算法为comp>>(AR2,BR1)，因为路径是串行操作，类似的，A2 → B2 → C1合并算法为comp>>(comp>>(AR2,BR2),CR1)。

最终得到的物理设备的规则集为：

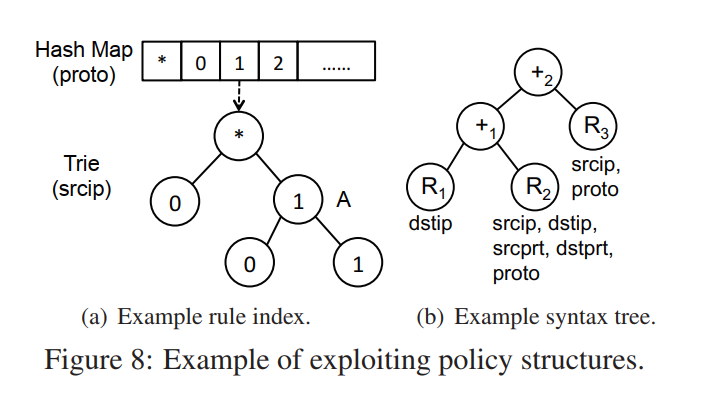




需要注意的是优先级的重新确定，在进行串行操作合并时，优先级的分配方式为comp>>的分配方式，也是就是，这种分配方式是错误的。SR3因为进行了三次合并，所以优先级更高，而实际上应该是SR2优先级更高，因此需要使用虚拟拓扑中虚拟交换机的数量作为指定长度，不足这个长度的优先级拼接需要补0，如。

# 五、效率优化

在组合两个规则集时，通常是一个n\*n的问题，方法复杂度为O(n2)，但是如果知道R1、R2的结构，就可以忽略一些结构不同的规则的组合，提高组合的速度。



在组合两个规则集时，可以对规则中的每个字段建立索引：

* 如果是精确匹配：建立hash索引
* 如果是前缀匹配：建立字典树索引
* 如果是通配符匹配：建立链表索引

此外，文章还考虑了对哪些字段进行索引，对于最直接相关的两个字段，那么需要索引的字段应该是：



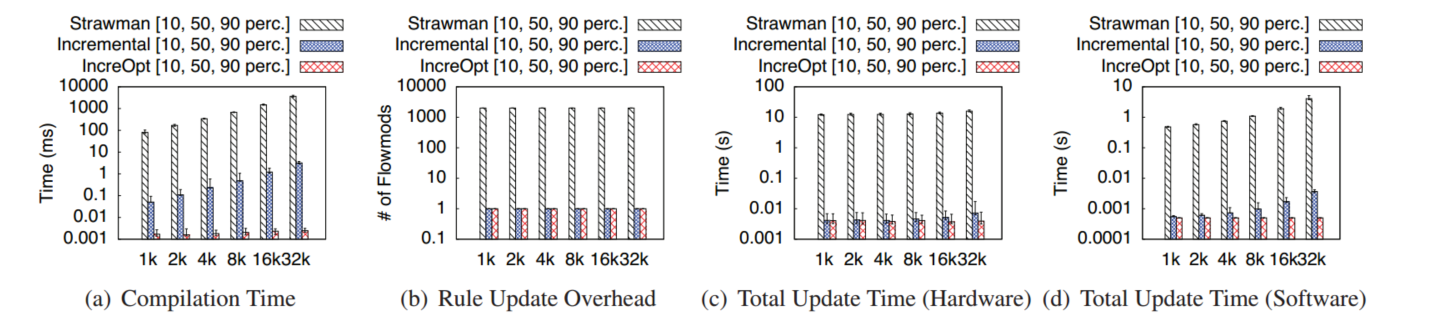
但是，如果当前合并的规则集Ri又是由其他规则集合并的，那么Ri的索引字段是：



因为至少需要一个字段来进行匹配。

# 六、实验结果

实验比较了Stawman方法（每次顺序更新大量优先级）、增量组合策略和优化后方案的编译时间（计算组合策略或编辑拓扑转换的时间）、规则更新负载（产生OpenFlow规则更新命令flowmod的条数）以及总的更新时间，实验效果如图所示：



FastUp：Fast TCAM

Flow Table Updates for SDN Switches

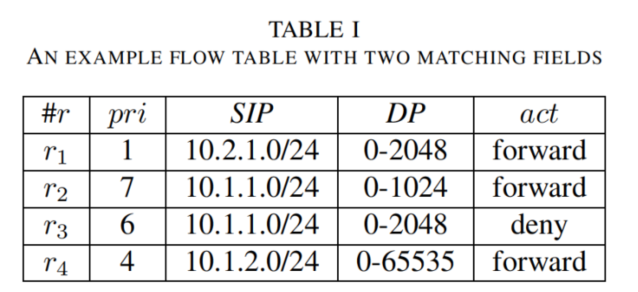
# 一、文章概述

更新TCAM流表分为两步，第一步是计算更新的过程，第二步是打断查询，应用更新步骤。现有的算法都只关注第二步的优化，而忽略了第一步的优化，此外部分算法忽略了优先级反转的问题，因此实际运行效率低于理论值。FastUp是一种新的高效的算法，极大地缩减了计算时间，并且打断查询的时间缩短到了10%。相比于其他的state-of-art TCAM流表更新算法，FastUp把更新延时缩短了10倍，把更新吞吐量提高了3-5倍。

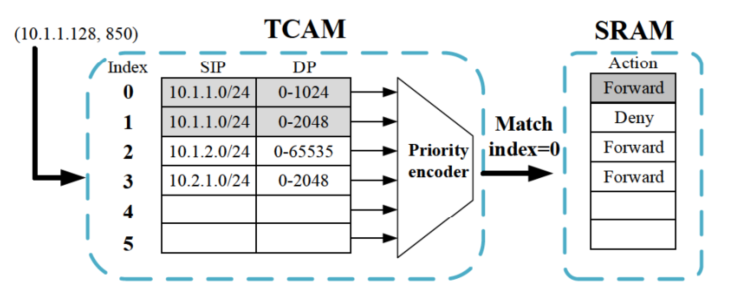
# 二、背景介绍

一个新的规则插入TCAM时通常分为两步，第一步是计算TCAM的更新步骤，时间为tc，第二步是打断TCAM查询过程，应用更新步骤，这一步的时间为tu。这些步骤代表了TCAM流表更新算法的两个优化方向：一方面需要减少流表更新对流表查询的影响，另一方面流表更新又要足够快，满足应用的需求。

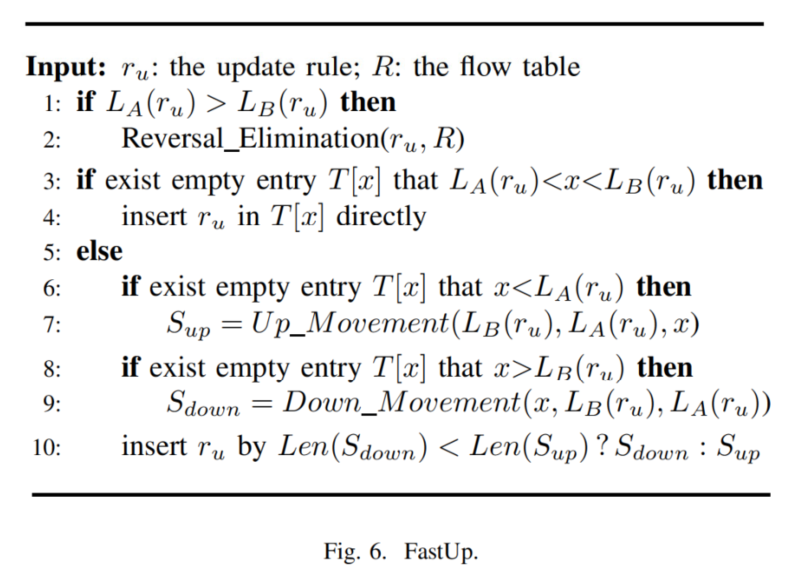
流表规则是一个由{pri, sp, act}组成的三元组，其中pri代表规则的优先级，sp代表需要匹配哪些项，act代表匹配成功后需要执行哪些动作。一个流表的示例如图所示：



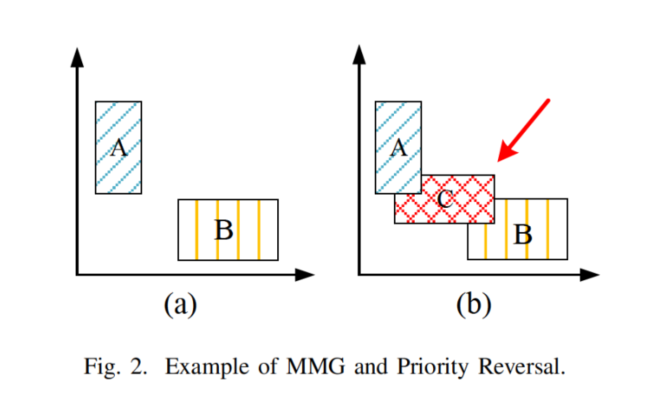
TCAM用来存储流表并进行查询，TCAM查询最适合的匹配项（通常是返回所有匹配规则的最低的index），并在并在相关联的SRAM中查找相应的action。优先级更高的表项通常存储在地址更低的位置，这样在第一次查询匹配时就直接返回匹配的索引。



# 三、FastUp算法设计



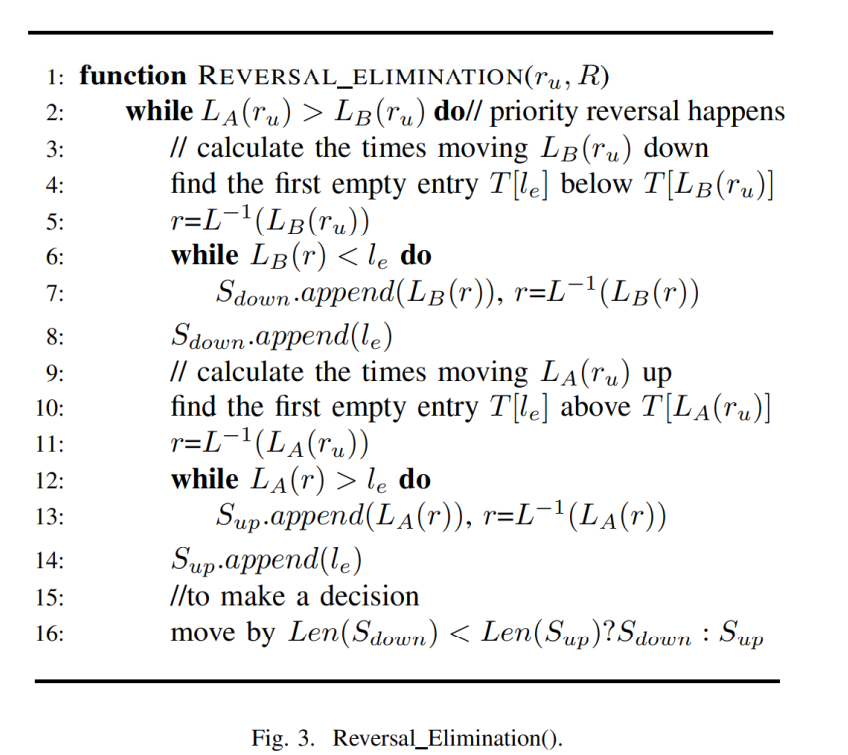
上图展示了算法的总体思路，第一步：消除优先级翻转的现象，找到一个一个可以插入的范围。第二步：在这个范围中找一个可以插入的位置，如果没有，则进行调整。



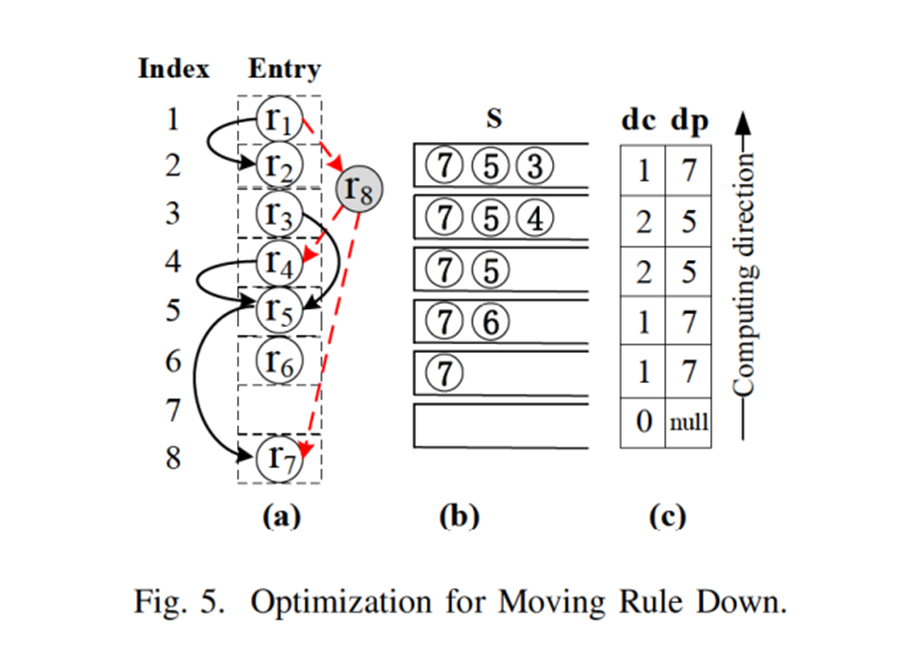
优先级翻转是指原来A和B之间没有依赖关系，所以在TCAM里随机排列，可能B就排在优先级更高的位置，但是插入规则C之后，A、B、C之间建立了依赖关系，A的优先级反而是最高的，新插入的C应该在B上面，在A下面，有可能在前一步中，B排在A的上面，这时就出现了优先级翻转，C找不到可以插入的位置范围。

调整优先级翻转的算法如下图所示：

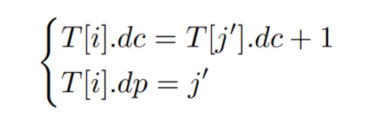
1. 找到下方的一个空格，判断这个空格上面有多少规则优先级低于新插入的规则，记这个数量为|Sdown|
2. 找到下方的一个空格，判断这个空格上面有多少规则优先级低于新插入的规则，记这个数量为|Sdown|
3. 比较|Sdown|与|Sup|，如果|Sup|更小，则移动Sup，否则移动Sdown
4. 重复这个过程，直到找到一个可以插入的范围。



在调整完优先级翻转的现象后，我们得到一个可以插入的范围，在这个范围内计算每个规则的调整开销cost，选择开销最小的进行调整。

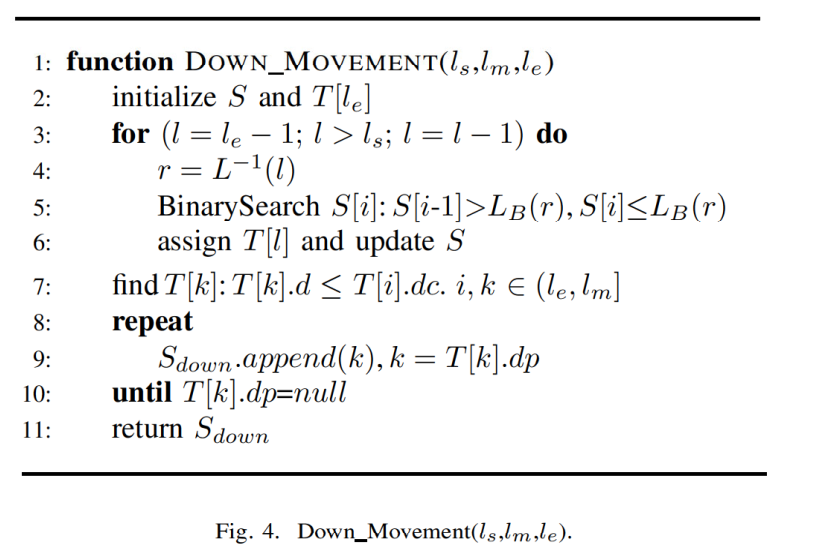


调整开销是指这个规则移动到下一个空位置需要移动的规则个数，用dc来表示。计算cost本质上是一个动态规划的过程：



但是，如果i<j并且T[i].dc<T[j].dc，那么在计算x<i的dc时就不需要考虑j的情况，基于这个规则，我们可以用额外的数据结构S来存储当前规则计算dc时需要考虑哪些规则。

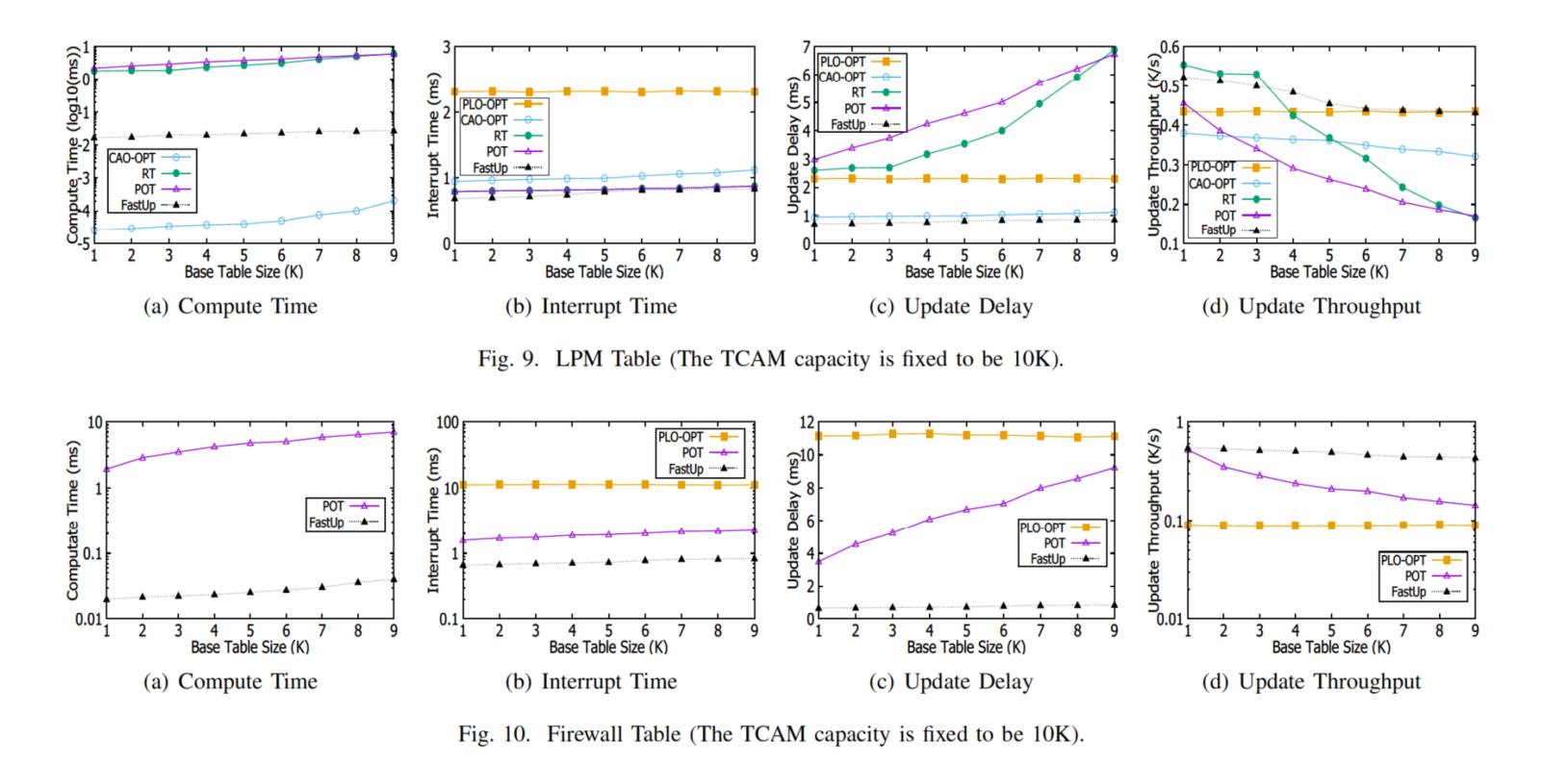
因此这一步完整的算法描述如下图所示：



由于S中需要关注规则的索引是有序的，所以可以使用二分搜索来查找需要关注的规则。在S中查找的复杂度为O(logk)，k是S中元素的个数，执行更新的复杂度为O(n)，因为上下界是之前算好的，所以获取上下界的复杂度是O(1)，最终算法复杂度是O(nlogk)，由于n远大于k，所以最终算法复杂度为O(n)。

# 四、实验结果

硬件为ONetSwitch，数据集为LPM和FireWal，度量计算时间、应用更新时间、更新延时和更新吞吐量。



在计算时间上，FastUp几乎保持常数的时间，与表的大小无关。在更新时间上，FastUp比PLO\_OPT快3倍，比POT快13倍，在更新延时上快3-8和8-10倍，并且保持稳定，在吞吐量上随着表的增大而稳定变化，而不像其他算法急剧减少。总之，FastUp在TCAM打断时间上已经接近最优。