# 对生命周期短、负载量少的流的处理方法

## 3.1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。

## 3.2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。

## 3.2 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。

### 3.2.1 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。

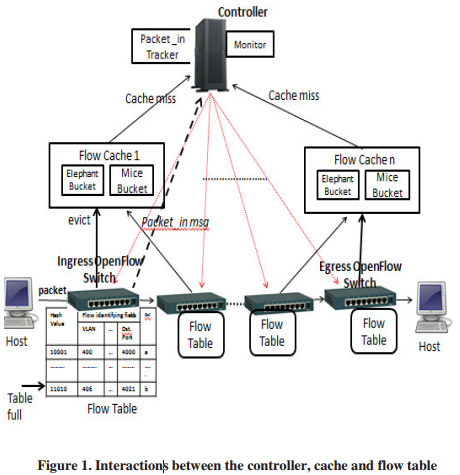


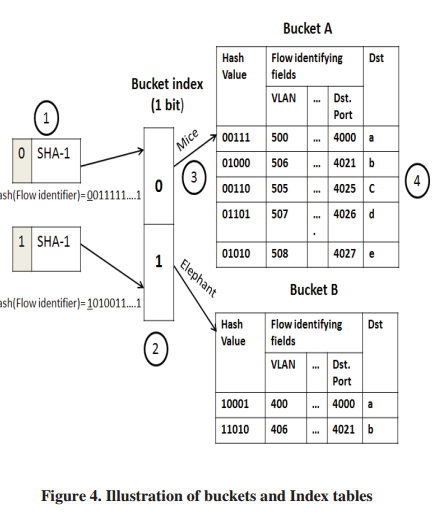
图1显示了OpenFlow交换机，流缓存和控制器之间的交互。 由于容量溢出或基于超时机制，将逐出流表中的条目。这些条目将存储在关联的流缓存中。当数据包到达启用OpenFlow的交换机时，首先在OpenFlow交换机的流表上执行查找，流表未命中将查找流缓存；如果流缓存中存在匹配，则执行相关的数据包转发操作；否则，packet\_in事件被发送到控制器并在流表中插入一个条目。如果缓存桶已满且最大桶数已经用完，则新的流条目会导致缓存未命中导致现有流的逐出。

### 3.2.2 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）。

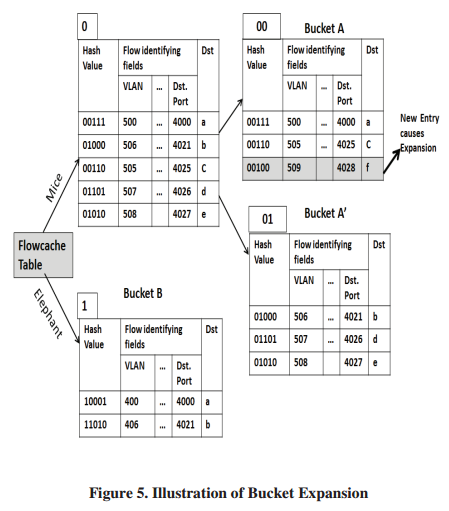


当一个桶的数量已经满了，而有一个新的条目要加入时，桶扩张机制（bucket expansion mechanism）将创建一个新桶。

### 3.2.3 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。



# 缓存策略问题

## 4.1 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。

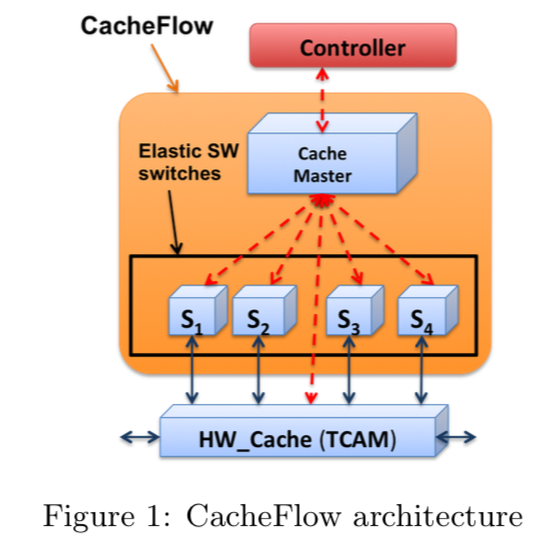
该CacheFlow原型实验证明规则拼接可以有效利用有限的TCAM空间，同时可以快速适应策略和流量需求的变化。

## 4.2 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。

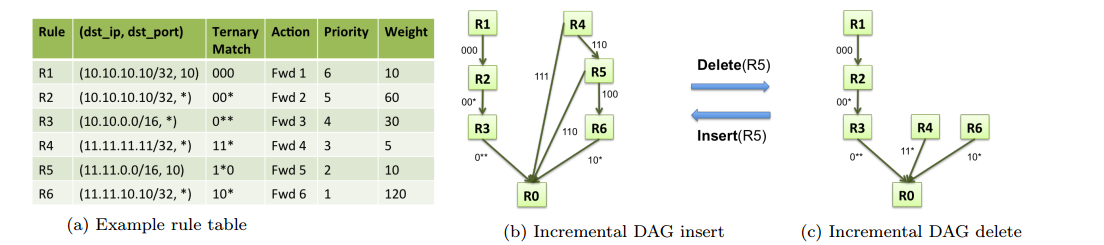
* CacheFlow由CacheMaster模块组成，该模块从SDN控制器接收OpenFlow命令。
* CacheMaster保留OpenFlow接口的语义，包括更新规则，查询计数器等的能力。
* CacheMaster使用OpenFlow协议将规则分发给未修改的商品硬件和软件交换机。
* CacheMaster是一个纯粹的控制平面组件，控制会话显示为虚线，数据平面转发显示为实线。

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。



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。

## 4.3 规则依赖的构建与更新



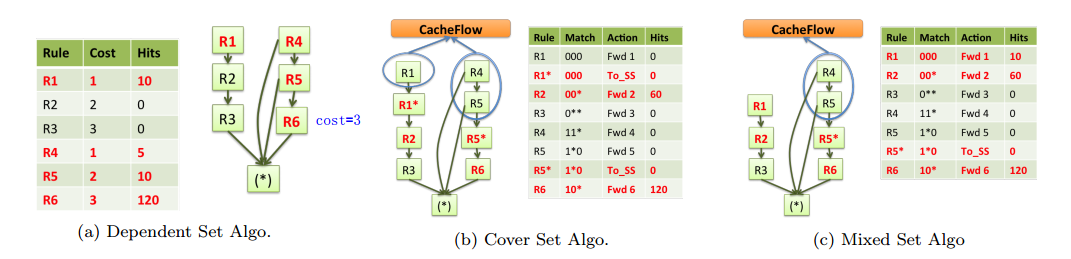
捕获规则表中所有依赖关系的简明方法是构造一个有向图，其中每个规则都是一个节点，每个边捕获一对规则之间的直接依赖关系，如图2（b）所示。 在以下条件下，子规则Ri和父规则Rj之间存在直接依赖关系：如果从规则表中删除Ri，则应该命中Ri的数据包现在将命中规则Rj。

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。

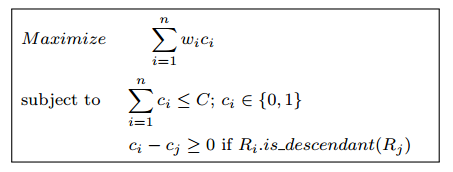
上述算法在更新时有着严重的效率问题，通过观察我们发现上图（b）我们发现，其实只用扫描左半边结点。因此提出了一种增量更新DAG图的算法。当更新一个结点时，不去打扰与它没有直接或间接依赖的其他结点。该算法时DAG图的更新速率得到了大幅度减少。

## 4.4 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。



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。



### 4.4.1 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。

### 4.4.2 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。

### 4.4.3 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。