CacheFlow: Dependency-Aware Rule-Caching for Software-Defined Networks

Naga Katta[1], Omid Alipourfard[2], Jennifer Rexford[1] and David Walker[1]

[1]Princeton University ({nkatta,jrex,dpw}@cs.princeton.edu)

[2]University of Southern California ({alipourf}@usc.edu)

ABSTRACT

Software-Defined Networking (SDN) allows control applications to install fine-grained forwarding policies in the underlying switches. While Ternary Content Addressable Memory (TCAM) enables fast lookups in hardware switches with flexible wildcard rule patterns, the cost and power requirements limit the number of rules the switches can support. To make matters worse, these hardware switches cannot sustain a high rate of updates to the rule table.

In this paper, we show how to give applications the illusion of high-speed forwarding, large rule tables, and fast updates by combining the best of hardware and software processing. Our CacheFlow system "caches" the most popular rules in the small TCAM, while relying on software to handle the small amount of "cache miss" traffic. However, we cannot blindly apply existing cache-replacement algorithms, because of dependencies between rules with overlapping patterns. Rather than cache large chains of dependent rules, we "splice" long dependency chains to cache smaller groups of rules while preserving the semantics of the policy. Experiments with our CacheFlow prototype—on both real and synthetic workloads and policies—demonstrate that rule splicing makes effective use of limited TCAM space, while adapting quickly to changes in the policy and the traffic demands.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Architecture and Design

Keywords

Rule Caching; Software-Defined Networking; OpenFlow; Commodity Switch; TCAM.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SOSR'16, March 14–15, 2016, Santa Clara, CA, USA

© 2016 ACM. ISBN 978-1-4503-4211-7/16/03…$15.00

DOI: http://dx.doi.org/10.1145/2890955.2890969

1. INTRODUCTION

In a Software-Defined Network (SDN), a logically centralized controller manages the flow of traffic by installing simple packet-processing rules in the underlying switches. These rules can match on a wide variety of packet-header fields, and perform simple actions such as forwarding, flooding, modifying the headers, and directing packets to the controller. This flexibility allows SDN-enabled switches to behave as firewalls, server load balancers, network address translators, Ethernet switches, routers, or anything in between. However, fine-grained forwarding policies lead to a large number of rules in the underlying switches.

In modern hardware switches, these rules are stored in Ternary Content Addressable Memory (TCAM) [1]. A TCAM can compare an incoming packet to the patterns in all of the rules at the same time, at line rate. However, commodity switches support relatively few rules, in the small thousands or tens of thousands [2]. Undoubtedly, future switches will support larger rule tables, but TCAMs still introduce a fundamental trade-off between rule-table size and other concerns like cost and power. TCAMs introduce around 100 times greater cost [3] and 100 times greater power consumption [4], compared to conventional RAM. Plus, updating the rules in TCAM is a slow process—today's hardware switches only support around 40 to 50 rule-table updates per second [5, 6], which could easily constrain a large network with dynamic policies.

Software switches may seem like an attractive alternative. Running on commodity servers, software switches can process packets at around 40 Gbps on a quad-core machine [7–10] and can store large rule tables in main memory and (to a lesser extent) in the L1 and L2 cache. In addition, software switches can update the rule table more than ten times faster than hardware switches [6]. But, supporting wildcard rules that match on many header fields is taxing for software switches, which must resort to slow processing (such as a linear scan) in user space to handle the first packet of each flow [7]. So, they cannot match the "horsepower" of hardware switches that provide hundreds of Gbps of packet processing (and high port density).

Fortunately, traffic tends to follow a Zipf distribution, where the vast majority of traffic matches a relatively small fraction of the rules [11]. Hence, we could leverage a small TCAM to forward the vast majority of traffic, and rely on software switches for the remaining traffic. For example, an 800 Gbps hardware switch, together with a single 40 Gbps software packet processor could easily handle traffic with a 5% "miss rate" in the TCAM. In addition, most rule-

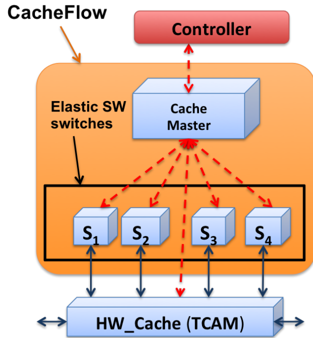


Figure 1: CacheFlow architecture

table updates could go to the slow-path components, while promoting very popular rules to hardware relatively infrequently. Together, the hardware and software processing would give controller applications the illusion of high-speed packet forwarding, large rule tables, and fast rule updates.

Our CacheFlow architecture consists of a TCAM and a *sharded* collection of software switches, as shown in Figure 1. The software switches can run on CPUs in the data plane (e.g., network processors), as part of the software agent on the hardware switch, or on separate servers. CacheFlow consists of a CacheMaster module that receives OpenFlow commands from an *unmodified* SDN controller. CacheMaster preserves the semantics of the OpenFlow interface, including the ability to update rules, query counters, etc. CacheMaster uses the OpenFlow protocol to distribute rules to *unmodified* commodity hardware and software switches. CacheMaster is a purely *control-plane* component, with control sessions shown as dashed lines and data-plane forwarding shown by solid lines.

As the name suggests, CacheFlow treats the TCAM as a *cache* that stores the most popular rules. However, we cannot simply apply existing cache-replacement algorithms, because the rules can match on overlapping sets of packets, leading to dependencies between multiple rules. Indeed, the switch we used for our experiments makes just this mistake (See §5)—a bug now addressed by our new system! Moreover, while earlier work on IP route caching [11–14] considered rule dependencies, IP prefixes only have simple "containment" relationships, rather than patterns that *partially* overlap. The partial overlaps can also lead to long dependency chains, and this problem is exacerbated by applications that combine multiple functions (like server load balancing and routing, as can be done in Frenetic [15] and CoVisor [16]) to generate many more rules.

To handle rule dependencies, we construct a compact representation of the given prioritized list of rules as an annotated directed acyclic graph (DAG), and design incremental algorithms for adding and removing rules to this data structure. Our cache-replacement algorithms use the DAG to decide which rules to place in the TCAM. To preserve rule-table space for the rules that match a large fraction of the traffic, we design a novel "splicing" technique that breaks long dependency chains. Splicing creates a few new rules that "cover" a large number of unpopular rules, to avoid polluting the cache. The technique extends to handle changes in the rules, as well as changes in their popularity over time.

In summary, we make the following key technical contributions:

- **Incremental rule-dependency analysis:** We develop an algorithm for incrementally analyzing and maintaining rule dependencies.
- **Novel cache-replacement strategies:** We develop new algorithms that only cache heavy-hitting rules along with a small set of dependencies.
- **Implementation and evaluation:** We discuss how CacheFlow preserves the semantics of the OpenFlow interface. Our experiments on both synthetic and real workloads show a cache-hit rate of 90% of the traffic by caching less than 5% of the rules.

A preliminary version of this work appeared as a workshop paper [17] which briefly discussed ideas about rule dependencies and caching algorithms. In this paper, we develop novel algorithms that help efficiently deal with practical deployment constraints like incremental updates to policies and high TCAM update times. We also evaluate CacheFlow by implementing large policies on actual hardware as opposed to simulations done using much smaller policies in the workshop version.

2. IDENTIFYING RULE DEPENDENCIES

In this section, we show how rule dependencies affect the correctness of rule-caching techniques and where such dependencies occur. We show how to represent cross-rule dependencies as a graph, and present efficient algorithms for incrementally computing the graph.

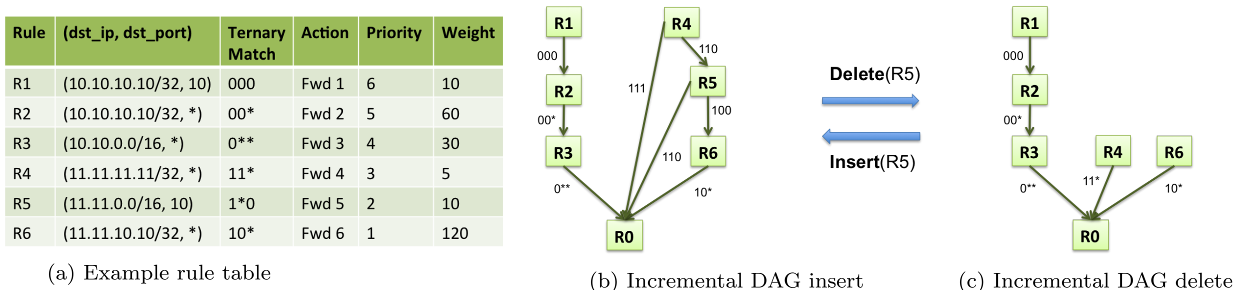
2.1 Rule Dependencies

The OpenFlow policy on a switch consists of a set of packet-processing rules. Each rule has a pattern, a priority, a set of actions, and counters. When a packet arrives, the switch identifies the highest-priority matching rules, performs the associated actions and increments the counters. CacheFlow implements these policies by splitting the set of rules into two groups—one residing in the TCAM and another in a software switch.

The semantics of CacheFlow is that (1) the highest-priority matching rule in the TCAM is applied, if such a rule exists, and (2) if no matching rule exists in the TCAM, then the highest-priority rule in the software switch is applied. As such, not all splits of the set of rules lead to valid implementations. If we do not cache rules in the TCAM *correctly*, packets that should hit rules in the software switch may instead hit a cached rule in the TCAM, leading to incorrect processing.

In particular, *dependencies* may exist between rules with differing priorities, as shown in the example in Figure 2(a). If the TCAM can store four rules, we cannot select the four rules with highest traffic volume (i.e., R_2, R_3, R_5, and R_6), because packets that should match R_1 (with pattern 000) would match R_2 (with pattern 00*); similarly, some packets (say with header 110) that should match R_4 would match R_5 (with pattern 1*0). That is, rules R_2 and R_5 *depend* on rules R_1 and R_4, respectively. In other words, there is a dependency from rule R_1 to R_2 and from rule R_4 to R_5. If R_2 is cached in the TCAM, R_1 should also be cached to preserve the semantics of the policy, similarly with R_5 and R_4.

A *direct* dependency exists between two rules if the patterns in the rules intersect (e.g., R_2 is dependent on R_1).



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

Figure 2: Constructing the rule dependency graph (edges annotated with reachable packets)

When a rule is cached in the TCAM, the corresponding dependent rules should also move to the TCAM. However, simply checking for intersecting patterns does *not* capture all of the policy dependencies. For example, going by this definition, the rule R_6 only depends on rule R_5. However, if the TCAM stored only R_5 and R_6, packets (with header 110) that should match R_4 would inadvertently match R_5 and hence would be incorrectly processed by the switch. In this case, R_6 also depends *indirectly* on R_4 (even though the matches of R_4 and R_6 do *not* intersect), because the match for R_4 overlaps with that of R_5. Therefore we need to define carefully what constitutes a dependency to handle such cases properly.

2.2 Where do complex dependencies arise?

Partial overlaps. Complex dependencies do not arise in traditional destination prefix forwarding because a prefix is dependent only on prefixes that are strict subsets of itself (nothing else). However, in an OpenFlow rule table, where rules have priorities and can match on multiple header fields, indirect dependencies occur because of partial overlaps between rules—both R_4 and R_6 only partially overlap with R_5. Hence, even though R_4 and R_6 do not have a direct dependency, they have an indirect dependency due to R_5's own dependence on R_6. The second column of Figure 2(a) illustrates such a situation. Here, one might interpret the first two bits of R_4, R_5, and R_6 as matching a destination IP, and the last bit as matching a port. Note that it is not possible to simply transform these rules into large FIB tables that are supported by today's switches because FIB lookups match on a single header field. Even if one were to somehow split a wild card rule into multiple FIB rules, it does not preserve counters.

Policy composition. Frenetic [15], Pyretic [18], CoVisor [16] and other high-level SDN programming platforms support abstractions for constructing complex network policies from a collection of simpler components. While the separate components may exhibit few dependencies, when they are compiled together, the composite rule tables may contain many complex dependencies. For instance, Figure 3(c) presents an illustrative rule table drawn from the CoVisor project [16]. The corresponding dependency graph for the rules is shown in Figure 3(d). Here, the dependency between rules R_3 and R_4 was not seen in the two separate components that defined the high-level policy, but does arise when they are composed.

Dependencies in REANNZ policies. We also analyzed a number of policies drawn from real networks to determine the nature of the dependencies they exhibit. As an example, Figure 3(a) shows part of an OF policy in use at the REANNZ research network [19]. The corresponding dependency graph can be seen in Figure 3(b). Now, one might conjecture that a network operator could manually rewrite the policy to reduce the number of dependencies and thereby facilitate caching. However, doing so is bound to be extremely tedious and highly error prone. Moreover, a good split may depend on the dynamic properties of network traffic. We argue that such tasks are much better left to algorithms, such as the ones we propose in this paper. An expert can develop a single caching algorithm, validate it and then deploy it on any policy. Such a solution is bound to be more reliable than asking operators to manually rewrite policies.

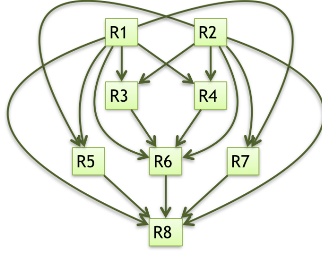
Is this a temporary problem? Even if the TCAM available on future switches grows, network operators will only become greedier in utilizing these resources—in the same way that with increasing amounts of DRAM, user applications have begun to consume increasing amounts of memory in conventional computers. Newer switches have multi-table pipelines [20] that can help avoid rule blowup from policy composition but the number of rules is still limited by the available TCAM. Our algorithms can be used to cache rules independently in each table (which maximizes the cache-hit traffic across all tables). Further, even high-end software switches like the OpenVSwitch (OVS) spend considerable effort [7] to cache popular OpenFlow rules in the kernel so that majority of the traffic does not get processed by the user-level classifier which is very slow. Thus, the rise of software switches may not completely avoid the problem of correctly splitting rule dependencies to cache them in the faster classifier process. Thus we believe our efforts are widely applicable and are going to be relevant in the long term despite the near term industry trends.

2.3 Constructing the Dependency DAG

A concise way to capture all the dependencies in a rule table is to construct a directed graph where each rule is a node, and each edge captures a direct dependency between a pair of rules as shown in Figure 2(b). A direct dependency exists between a child rule R_i and a parent rule R_j under the following condition—if R_i is removed from the rule table, packets that are supposed to hit R_i will now hit rule R_j. The edge between the rules in the graph is annotated by the set of packets that reach the parent from the child. Then, the dependencies of a rule consist of all descendants of that rule (e.g., R_1 and R_2 are the dependencies for R_3). The rule R_0 is the default *match-all* rule (matches all packets with priority 0) added to maintain a connected rooted graph without altering the overall policy.

Rule	Priority	Match
R1	32800	tcp;dport:179
R2	32800	tcp;dport:646
R3	16640	dstip:111.221.69.0/24
R4	16640	dstip:111.221.66.0/24
R5	16630	dstip:111.221.78.0/23
R6	16610	dstip:111.221.64.0/21
R7	16610	dstip:111.221.112.0/21
R8	16580	dstip:111.221.64.0/18

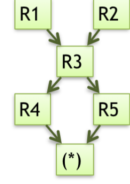
(a) Reanzz Rule Table



(b) Reanzz Subgraph

Rule	Priority	Match
R1	5	srcip = 1.0.0.0/24 dstip = 2.0.0.1
R2	4	srcip = 1.0.0.0/24 dstip = 2.0.0.2
R3	3	srcip = 1.0.0.0/24
R4	2	dstip = 2.0.0.1
R5	1	dstip = 2.0.0.2

(c) CoVisor Example Table



(d) CoVisor Example Graph

Figure 3: Dependent-set vs. cover-set algorithms (L_0 cache rules in red)

Algorithm 1: Building the dependency graph

```
   // Add dependency edges
1  func addParents(R:Rule, P:Parents) begin
2      deps = (∅);
       // p.o : priority order
3      packets = R.match;
4      for each R_j in P in descending p.o: do
5          if (packets ∩ R_j.match) != ∅ then
6              deps = deps ∪ {(R,R_j)};
7              reaches(R,R_j) = packets ∩ R_j;
8              packets = packets - R_j.match;
9      return deps;
10 for each R:Rule in Pol:Policy do
11     potentialParents = [R_j in Pol | R_j.p.o ≤ R.p.o];
12     addParentEdges(R, potentialParents);
```

To identify the edges in the graph, for any given child rule R, we need to find out all the parent rules that the packets matching R can reach. This can be done by taking the symbolic set of packets matching R and iterating them through all of the rules with lower priority than R that the packets might hit.

To find the rules that depend directly on R, Algorithm 1 scans the rules R_i with lower priority than R (line 14) in order of decreasing priority. The algorithm keeps track of the set of packets that can reach each successive rule (the variable `packets`). For each such new rule, it determines whether the predicate associated with that rule intersects[1] the set of packets that can reach that rule (line 5). If it does, there is a dependency. The arrow in the dependency edge points from the child R to the parent R_i. In line 7, the dependency edge also stores the packet space that actually reaches the parent R_i. In line 8, before searching for the next parent, because the rule R_i will now occlude some packets from the current `reaches` set, we subtract R_i's predicate from it.

This compact data structure captures *all* dependencies because we track the flow of all the packets that are processed by any rule in the rule table. The data structure is a directed acyclic graph (DAG) because if there is an edge from R_i to R_j then the priority of R_i is always strictly greater than priority of R_j. Note that the DAG described here is *not* a topological sort (we are not imposing a total order on vertices of a graph but are computing the edges themselves). Once such a dependency graph is constructed, if a rule R is to be cached in the TCAM, then all the descendants of R in the dependency graph should also be cached for correctness.

[1]Symbolic intersection and subtraction of packets can be done using existing techniques [21].

2.4 Incrementally Updating The DAG

Algorithm 1 runs in $\mathcal{O}(n^2)$ time where n is the number of rules. As we show in Section 6, running the static algorithm on a real policy with 180K rules takes around 15 minutes, which is unacceptable if the network needs to push a rule into the switches as quickly as possible (say, to mitigate a DDoS attack). Hence we describe an incremental algorithm that has considerably smaller running time in most practical scenarios—just a few milliseconds for the policy with 180K rules.

Figure 2(b) shows the changes in the dependency graph when the rule R_5 is inserted. All the changes occur only in the right half of the DAG because the left half is not affected by the packets that hit the new rule. A rule insertion results in three sets of updates to the DAG: (i) existing dependencies (like (R_4,R_0)) change because packets defining an existing dependency are impacted by the newly inserted rule, (ii) creation of dependencies with the new rule as the parent (like (R_4,R_5)) because packets from old rules (R_4) are now hitting the new rule (R_5), and (iii) creation of dependencies (like (R_5,R_6)) because the packets from the new rule (R_5) are now hitting an old rule (R_6). Algorithm 1 takes care of all three dependencies by it rebuilding *all* dependencies from scratch. The challenge for the incremental algorithm is to do the same set of updates without touching the irrelevant parts of the DAG — In the example, the left half of the DAG is not affected by packets that hit the newly inserted rule.

2.4.1 *Incremental Insert*

In the incremental algorithm, the intuition is to use the `reaches` variable (packets reaching the parent from the child) cached for each existing edge to recursively traverse only the necessary edges that need to be updated. Algorithm 2 proceeds in three phases:

(i) Updating existing edges (lines 1–10): While finding the affected edges, the algorithm recursively traverses the dependency graph beginning with the default rule R_0. It checks if the newRule intersects any edge between the current node and its children. It updates the intersecting edge and adds it to the set of affected edges (line 4). However, if newRule is higher in the priority chain, then the recursion proceeds exploring the edges of the next level (line 9). It also collects the rules that could potentially be the parents as it climbs up the graph (line 8). This way, we end up only exploring the relevant edges and rules in the graph.

Algorithm 2: Incremental DAG insert

```
1  func FindAffectedEdges(rule, newRule) begin
2      for each C in Children(rule) do
3          if Priority(C) > priority(newRule) then
4              if reaches(C,rule) ∩ newRule.match != ∅
               then
5                  reaches(C, rule) -= newRule.match;
6                  add (C, Node) to affEdges
7          else
8              if Pred(C) ∪ newRule.match != ∅ then
9                  add C to potentialParents;
10                 FindAffectedEdges(C, newRule);
11 func processAffectedEdges(affEdges) begin
12     for each childList in groupByChild(affEdges) do
13         deps = deps ∪ {(child, newRule)};
14         edgeList = sortByParent(childList);
15         reaches(child, newRule) = reaches(edgeList[0]);
16 func Insert(G=(V, E), newNode) begin
17     affEdges = { };
18     potentialParents = [R_0];
19     FindAffectedEdges(R_0, newNode);
20     ProcessAffectedEdges(affEdges);
21     addParents(newNode, potentialParents);
```

(ii) Adding directly dependent children (lines 11-15): In the second phase, the set of affected edges collected in the first phase are grouped by their children. For each child, an edge is created from the child to the newRule using the packets from the child that used to reach its highest priority parent (line 14). Thus all the edges from the new rule to its children are created.

(iii) Adding directly dependent parents (line 21): In the third phase, all the edges that have newRule as the child are created using the `addParents` method described in Algorithm 1 on all the potential parents collected in the first phase.

In terms of the example, in phase 1, the edge (R_4, R_0) is the affected edge and is updated with `reaches` that is equal to 111 (11* - 1*0). The rules R_0 and R_6 are added to the new rule's potential parents. In phase 2, the edge (R_4, R_5) is created. In phase 3, the function `addParents` is executed on parents R_6 and R_0. This results in the creation of edges (R_5, R_6) and (R_5, R_0).

Running Time: Algorithm 2 clearly avoids traversing the left half of the graph which is not relevant to the new rule. While in the worst case, the running time is linear in the number of edges in the graph, for most practical policies, the running time is linear in the number of closely related dependency groups[2].

2.4.2 Incremental Delete

The deletion of a rule leads to three sets of updates to a dependency graph: (i) new edges are created between other rules whose packets used to hit the removed rule, (ii) existing edges are updated because more packets are reaching this dependency because of the absence of the removed rule, and

[2]Since the dependency graph usually has a wide bush of isolated prefix dependency chains—like the left half and right half in the example DAG—which makes the insertion cost equal to the number of such chains.

Algorithm 3: Incremental DAG delete

```
1  func Delete(G=(V, E), oldRule) begin
2      for each c in Children(oldRule) do
3          potentialParents = Parents(c) - {oldRule};
4          for each p in Parents(oldRule) do
5              if reaches(c, oldRule) ∩ p.match != ∅ then
6                  add p to potentialParents
7          addParents(C, potentialParents)
8      Remove all edges involving oldRule
```

(iii) finally, old edges having the removed rule as a direct dependency are deleted.

For the example shown in Figure 2(c), where the rule R_5 is deleted from the DAG, existing edges (like (R_4, R_0)) are updated and all three involving R_5 are created. In this example, however, no new edge is created. But it is potentially possible in other cases (consider the case where rule R_2 is deleted which would result in a new edge between R_1 and R_3).

An important observation is that unlike an incremental insertion (where we recursively traverse the DAG beginning with R_0), incremental deletion of a rule can be done local to the rule being removed. This is because all three sets of updates involve only the children or parents of the removed rule. For example, a new edge can only be created between a child and a parent of the removed rule[3].

Algorithm 3 incrementally updates the graph when a new rule is deleted. First, in lines 2-6, the algorithm checks if there is a new edge possible between any child-parent pair by checking whether the packets on the edge (child, oldRule) reach any parent of oldRule (line 5). Second, in lines 3 and 7, the algorithm also collects the parents of all the existing edges that may have to be updated (line 3). It finally constructs the new set of edges by running the `addParents` method described in Algorithm 1 to find the exact edges between the child c and its parents (line 7). Third, in line 8, the rules involving the removed rule as either a parent or a child are removed from the DAG.

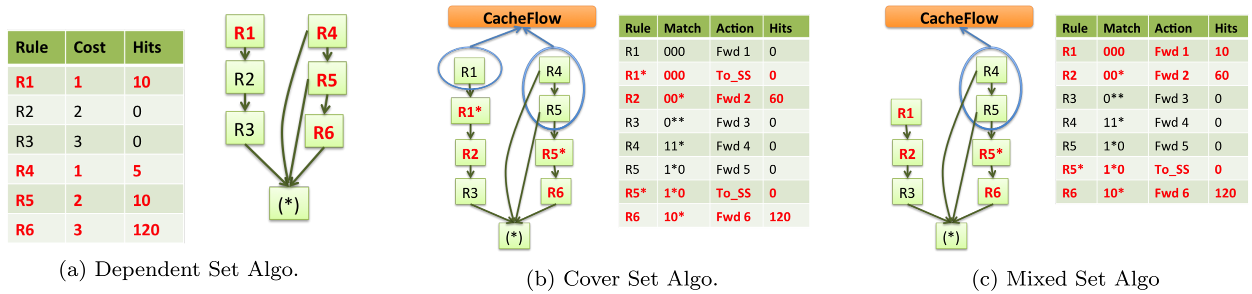
Running time: This algorithm is dominated by the two `for` loops (in lines 2 and 4) and may also have a worst case $\mathcal{O}(n^2)$ running time (where n is the number of rules) but in most practical policy scenarios, the running time is much smaller (owing to the small number of children/parents for any given rule in the DAG).

3. CACHING ALGORITHMS

In this section, we present CacheFlow's algorithm for placing rules in a TCAM with limited space. CacheFlow selects a set of important rules from among the rules given by the controller to be cached in the TCAM, while redirecting the cache misses to the software switches.

We first present a simple strawman algorithm to build intuition, and then present new algorithms that avoids caching low-weight rules. Each rule is assigned a "cost" corresponding to the number of rules that must be installed together and a "weight" corresponding to the number of packets ex-

[3]A formal proof is omitted for lack of space and is left for the reader to verify. In the example where R_2 is deleted, a new rule can only appear between R_1 and R_3. Similarly when R_5 is deleted, a new rule could have appeared between R_4 and R_6 but does not because the rules do not overlap.



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

Figure 4: Dependent-set vs. cover-set algorithms (L_0 cache rules in red)

pected to hit that rule [4]. Continuing with the running example from the previous section, R_6 depends on R_4 and R_5, leading to a cost of 3, as shown in Figure 4(a). In this situation, R_2 and R_6 hold the majority of the weight, but cannot be installed simultaneously on a TCAM with capacity 4, as installing R_6 has a cost of 3 and R_2 bears a cost of 2. Hence together they do not fit. The best we can do is to install rules R_1, R_4, R_5, and R_6 which maximizes total weight, subject to respecting all dependencies.

3.1 Optimization: NP Hardness

The input to the rule-caching problem is a dependency graph of n rules $R_1, R_2, \ldots, R_n$, where rule R_i has higher priority than rule R_j for $i < j$. Each rule has a match and action, and a weight w_i that captures the volume of traffic matching the rule. There are dependency edges between pairs of rules as defined in the previous section. The output is a prioritized list of C rules to store in the TCAM[5]. The objective is to maximize the sum of the weights for traffic that "hits" in the TCAM, while processing "hit" packets according to the semantics of the original rule table.

$$
\begin{aligned}
&\textit{Maximize} && \sum_{i=1}^{n} w_i c_i \\
&\text{subject to} && \sum_{i=1}^{n} c_i \leq C;\ c_i \in \{0,1\} \\
& && c_i - c_j \geq 0 \text{ if } R_i.is_descendant(R_j)
\end{aligned}
$$

The above optimization problem is NP-hard in n and k. It can be reduced from the densest k-subgraph problem which is known to be NP-hard. We outline a sketch of the reduction here between the decision versions of the two problems. Consider the decision problem for the caching problem: Is there a subset of C rules from the rule table which respect the directed dependencies and have a combined weight of atleast W. The decision problem for the densest k-subgraph problem is to ask if there is a subgraph incident on k vertices that has at least d edges in a given undirected graph G=(V,E) (This generalizes the well known CLIQUE problem for $d=\binom{k}{2}$, hence is hard).

[4]In practice, weights for rules are updated in an online fashion based on the packet count in a sliding window of time.
[5]Note that CacheFlow does *not* simply install rules on a cache miss. Instead, CacheFlow makes decisions based on traffic measurements over the recent past. This is important to defend against cache-thrashing attacks where an adversary generates low-volume traffic spread across the rules.

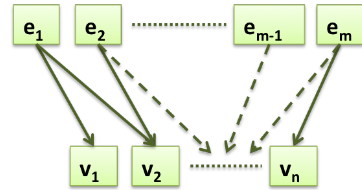


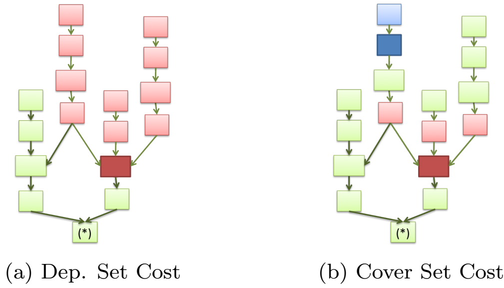
Figure 5: Reduction from densest k-subgraph

Consider the reduction shown in Figure 5. For a given instance of the densest k-subgraph problem with parameters k and d, we construct an instance of the cache-optimization problem in the following manner. Let the vertices of G' be nodes indexed by the vertices and edges of G. The edges of G' are constructed as follows: for every undirected edge $e = (v_i, v_j)$ in G, there is a directed edge from e to v_i and v_j. This way, if e is chosen to include in the cache, v_i and v_j should also be chosen. Now we assign weights to nodes in V' as follows : $w(v) = 1$ for all $v \in V$ and $w(e) = n + 1$ for all $e \in E$. Now let $C = k + d$ and $W = d(n + 1)$. If you can solve this instance of the cache optimization problem, then you have to choose at least d of the edges $e \in E$ because you cannot reach the weight threshold with less than d edge nodes (since their weight is much larger than nodes indexed by V). Since C cannot exceed $d+k$, because of dependencies, one will also end up choosing less than k vertices $v \in V$ to include in the cache. Thus this will solve the densest k-subgraph instance.

3.2 Dependent-Set: Caching Dependent Rules

No polynomial time approximation scheme (PTAS) is known yet for the densest k-subgraph problem. It is also not clear whether a PTAS for our optimization problem can be derived directly from a PTAS for the densest subgraph problem. Hence, we use a heuristic that is modeled on a greedy PTAS for the Budgeted Maximum Coverage problem [22] which is similar to the formulation of our problem. In our greedy heuristic, at each stage, the algorithm chooses a set of rules that maximizes the ratio of combined rule weight to combined rule cost ($\frac{\Delta W}{\Delta C}$), until the total cost reaches k. This algorithm runs in $O(nk)$ time.

On the example rule table in Figure 4(a), the greedy algorithm selects R_6 first (and its dependent set $\{R_4, R_5\}$), and then R_1 which brings the total cost to 4. Thus the set of rules in the TCAM are R_1, R_4, R_5, and R_6 which is the optimal. We refer to this algorithm as the *dependent-set* algorithm.



(a) Dep. Set Cost (b) Cover Set Cost

Figure 6: Dependent-set vs. cover-set Cost

3.3 Cover-Set: Splicing Dependency Chains

Respecting rule dependencies can lead to high costs, especially if a high-weight rule depends on many low-weight rules. For example, consider a firewall that has a single low-priority "accept" rule that depends on many high-priority "deny" rules that match relatively little traffic. Caching the one "accept" rule would require caching many "deny" rules. We can do better than past algorithms by modifying the rules in various semantics-preserving ways, instead of simply packing the existing rules into the available space—this is the key observation that leads to our superior algorithm. In particular, we "splice" the dependency chain by creating a small number of new rules that *cover* many low-weight rules and send the affected packets to the software switch.

For the example in Figure 4(a), instead of selecting all dependent rules for R_6, we calculate new rules that cover the packets that would otherwise incorrectly hit R_6. The extra rules direct these packets to the software switches, thereby breaking the dependency chain. For example, we can install a high-priority rule R_5^* with match 1*1* and action `forward_to_Soft_switch`,[6] along with the low-priority rule R_6. Similarly, we can create a new rule R_1^* to break dependencies on R_2. We avoid installing higher-priority, low-weight rules like R_4, and instead have the high-weight rules R_2 and R_6 inhabit the cache simultaneously, as shown in Figure 4(b).

More generally, the algorithm must calculate the *cover set* for each rule R. To do so, we find the immediate ancestors of R in the dependency graph and replace the actions in these rules with a `forward_to_Soft_Switch` action. For example, the cover set for rule R_6 is the rule R_5^* in Figure 4(b); similarly, R_1^* is the cover set for R_2. The rules defining these `forward_to_Soft_switch` actions may also be merged, if necessary.[7] The cardinality of the cover set defines the new cost value for each chosen rule. This new cost is strictly less than or equal to the cost in the dependent set algorithm. The new cost value is *much* less for rules with long chains of dependencies. For example, the old dependent set cost for the rule R_6 in Figure 4(a) is 3 as shown in the rule cost table whereas the cost for the new cover set for R_6 in Figure 4(b) is only 2 since we only need to cache R_5^* and R_6. To take a more general case, the old cost for the red rule in Figure 6(a) was the entire set of ancestors (in light red), but the new cost (in Figure 6(b)) is defined just by the immediate ancestors (in light red).

[6]This is just a standard forwarding action out some port connected to a software switch.

[7]To preserve OpenFlow semantics pertaining to hardware packet counters, policy rules cannot be compressed. However, we can compress the intermediary rules used for forwarding cache misses, since the software switch can track the per-rule traffic counters.

3.4 Mixed-Set: An Optimal Mixture

Despite decreasing the cost of caching a rule, the cover-set algorithm may also decrease the weight by redirecting the spliced traffic to the software switch. For example, for caching the rule R_2 in Figure 4(c), the dependent-set algorithm is a better choice because the traffic volume processed by the dependent set in the TCAM is higher, while the cost is the same as a cover set. In general, as shown in Figure 6(b), cover set seems to be a better choice for caching a higher dependency rule (like the red node) compared to a lower dependency rule (like the blue node).

In order to deal with cases where one algorithm may do better than the other, we designed a heuristic that chooses the best of the two alternatives at each iteration. As such, we consider a metric that chooses the *best* of the two sets i.e., $\max(\frac{\Delta W_{dep}}{\Delta C_{dep}}, \frac{\Delta W_{cover}}{\Delta C_{cover}})$. Then we can apply the same greedy covering algorithm with this new metric to choose the best set of candidate rules to cache. We refer to this version as the *mixed-set* algorithm.

3.5 Updating the TCAM Incrementally

As the traffic distribution over the rules changes over time, the set of cached rules chosen by our caching algorithms also change. This would mean periodically updating the TCAM with a new version of the policy cache. Simply deleting the old cache and inserting the new cache from scratch is not an option because of the enormous TCAM rule insertion time. It is important to minimize the churn in the TCAM when we periodically update the cached rules.

Updating just the difference will not work.

Simply taking the difference between the two sets of cached rules—and replacing the stale rules in the TCAM with new rules (while retaining the common set of rules)—can result in incorrect policy snapshots on the TCAM during the transition. This is mainly because TCAM rule update takes time and hence packets can be processed incorrectly by an incomplete policy snapshot during transition. For example, consider the case where the mixed-set algorithm decides to change the cover-set of rule R_6 to its dependent set. If we simply remove the cover rule (R_5^*) and then install the dependent rules (R_5,R_4), there will be a time period when only the rule R_6 is in the TCAM without either its cover rules or the dependent rules. This is a policy snapshot that can incorrectly process packets while the transition is going on.

Exploiting composition of mixed sets.

A key property of the algorithms discussed so far is that each chosen rule along with its mixed (cover or dependent) set can be added/removed from the TCAM independently of the rest of the rules. In other words, the mixed-sets for any two rules are easily composable and decomposable. For example, in Figure 6(b), the red rule and its cover set can be easily added/removed without disturbing the blue rule and its dependent set. In order to push the new cache in to the TCAM, we first decompose/remove the old mixed-sets (that are not cached anymore) from the TCAM and then compose the TCAM with the new mixed sets. We also maintain reference counts from various mixed sets to the rules on TCAM so that we can track rules in overlapping mixed sets. Composing two candidate rules to build a cache would simply involve merging their corresponding mixed-sets (and incre-

menting appropriate reference counters for each rule) and decomposing would involve checking the reference counters before removing a rule from the TCAM [8]. In the example discussed above, if we want to change the cover-set of rule R_6 to its dependent set on the TCAM, we first delete the entire cover-set rules (including rule R_6) and then install the entire dependent-set of R_6, in priority order.

4. CACHEMASTER DESIGN

As shown in Figure 1, CacheFlow has a CacheMaster module that implements its control-plane logic. In this section, we describe how CacheMaster directs "cache-miss" packets from the TCAM to the software switches, using existing switch mechanisms and preserves the semantics of OpenFlow.

4.1 Scalable Processing of Cache Misses

CacheMaster runs the algorithms in Section 3 to compute the rules to cache in the TCAM. The cache misses are sent to one of the software switches, which each store a copy of the entire policy. CacheMaster can shard the cache-miss load over the software switches.

Using the group tables in OpenFlow 1.1+, the hardware switch can apply a simple load-balancing policy. Thus the `forward_to_SW_switch` action (used in Figure 4) forwards the cache-miss traffic—say, matching a low-priority "catch-all" rule—to this load-balancing group table in the switch pipeline, whereupon the cache-miss traffic can be distributed over the software switches.

4.2 Preserving OpenFlow Semantics

To work with unmodified controllers and switches, CacheFlow preserves semantics of the OpenFlow interface, including rule priorities and counters, as well as features like `packet_in`s, barriers, and rule timeouts.

Preserving inports and outports: CacheMaster installs three kinds of rules in the hardware switch: (i) fine-grained rules that apply the cached part of the policy (cache-hit rules), (ii) coarse-grained rules that forward packets to a software switch (cache-miss rules), and (iii) coarse-grained rules that handle return traffic from the software switches, similar to mechanisms used in DIFANE [23]. In addition to matching on packet-header fields, an OpenFlow policy may match on the inport where the packet arrives. Therefore, the hardware switch *tags* cache-miss packets with the input port (e.g., using a VLAN tag) so that the software switches can apply rules that depend on the inport[9]. The rules in the software switches apply any "drop" or "modify" actions, tag the packets for proper forwarding at the hardware switch, and direct the packet back to the hardware switch. Upon receiving the return packet, the hardware switch simply matches

[8]The intuition is that if a rule has a positive reference count, then either its dependent-set or the cover-set is also present on the TCAM and hence is safe to leave behind during the decomposition phase

[9]Tagging the cache-miss packets with the inport can lead to extra rules in the hardware switch. In several practical settings, the extra rules are not necessary. For example, in a switch used only for layer-3 processing, the destination MAC address uniquely identifies the input port, obviating the need for a separate tag. Newer version of OpenFlow support switches with multiple stages of tables, allowing us to use one table to push the tag and another to apply the (cached) policy.

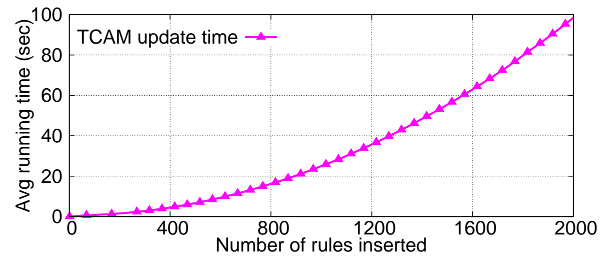


Figure 7: TCAM Update Time

on the tag, pops the tag, and forwards to the designated output port(s).

Packet-in messages: If a rule in the TCAM has an action that sends the packet to the controller, CacheMaster simply forwards the the `packet_in` message to the controller. However, for rules on the software switch, CacheMaster must transform the `packet_in` message by (i) copying the inport from the packet tag into the inport field of the `packet_in` message and (ii) stripping the tag from the packet before sending to the controller.

Traffic counts, barrier messages, and rule timeouts: CacheFlow preserves the semantics of OpenFlow constructs like queries on traffic statistics, barrier messages, and rule timeouts by having CacheMaster emulate these features. For example, CacheMaster maintains packet and byte counts for each rule installed by the controller, updating its local information each time a rule moves to a different part of the "cache hierarchy." The CacheMaster maintains three counters per rule. A hardware counter periodically polls and maintains the current TCAM counter for the rule if it cached. Similarly, a software counter maintains the current software switch counters. A persistent hardware count accumulates the hardware counter whenever the rule is removed from the hardware cache and resets the hardware counter to zero. Thus, when an application asks for a rule counter, CacheMaster simply returns the sum of the three counters associated with that rule.

Similarly, CacheMaster emulates [24] rule timeouts by installing rules *without* timeouts, and explicitly removing the rules when the software timeout expires. For barrier messages, CacheMaster first sends a barrier request to all the switches, and waits for all of them to respond before sending a barrier reply to the controller. In the meantime, CacheMaster buffers all messages from the controller before distributing them among the switches.

5. COMMODITY SWITCH AS THE CACHE

The hardware switch used as a cache in our system is a Pronto-Pica8 3290 switch running PicOS 2.1.3 supporting OpenFlow. We uncovered several limitations of the switch that we had to address in our experiments:

Incorrect handling of large rule tables: The switch has an ASIC that can hold 2000 OpenFlow rules. If more than 2000 rules are sent to the switch, 2000 of the rules are installed in the TCAM and the rest in the software agent. However, the switch does not respect the cross-rule dependencies when updating the TCAM, leading to incorrect forwarding behavior! Since we cannot modify the (proprietary) software agent, we simply avoid triggering this bug by assuming the rule capacity is limited to 2000 rules. Inter-

estingly, the techniques presented in this paper are exactly what the software agent can use to fix this issue.

Slow processing of control commands: The switch is slow at updating the TCAM and querying the traffic counters. The time required to update the TCAM is a non-linear function of the number of rules being added or deleted, as shown in Figure 7. While the first 500 rules take 6 seconds to add, the next 1500 rules takes almost 2 minutes to install. During this time, querying the switch counters easily led to the switch CPU hitting 100% utilization and, subsequently, to the switch disconnecting from the controller. In order to get around this, we wait till the set of installed rules is relatively stable to start querying the counters at regular intervals and rely on counters in the software switch in the meantime.

6. PROTOTYPE AND EVALUATION

We implemented a prototype of CacheFlow in Python using the Ryu controller library so that it speaks OpenFlow to the switches. On the north side, CacheFlow provides an interface which control applications can use to send `FlowMods` to CacheFlow, which then distributes them to the switches. At the moment, our prototype supports the semantics of the OpenFlow 1.0 features mentioned earlier (except for rule timeouts) transparently to both the control applications and the switches.

We use the Pica8 switch as the hardware cache, connected to an Open vSwitch 2.1.2 multithread software switch running on an AMD 8-core machine with 6GB RAM. To generate data traffic, we connected two host machines to the Pica8 switch and use `tcpreplay` to send packets from one host to the other.

6.1 Cache-hit Rate

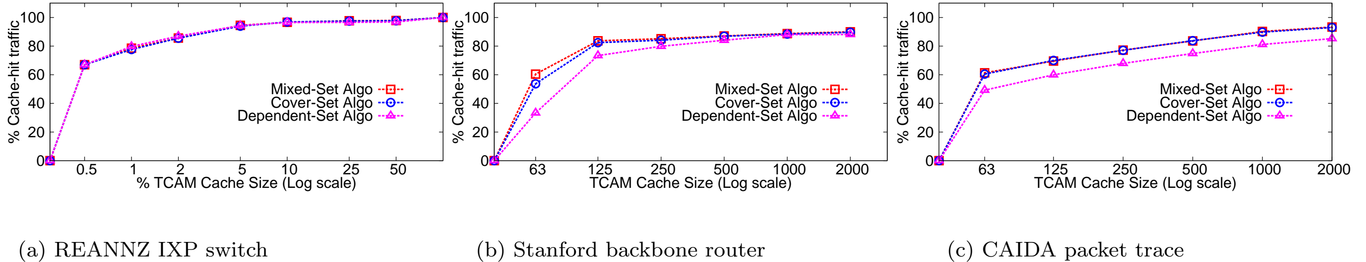
We evaluate our prototype against three policies and their corresponding packet traces: (i) A publicly available packet trace from a real data center and a synthetic policy, (ii) An educational campus network routing policy and a synthetic packet trace, and (iii) a real OpenFlow policy and the corresponding packet trace from an Internet eXchange Point (IXP). We measure the cache-hit rate achieved on these policies using three caching algorithms (dependent-set, cover-set, and mixed-set). The cache misses are measured by using `ifconfig` on the software switch port and then the cache hits are calculated by subtracting the cache misses from the total packets sent as reported by `tcpreplay`. All the results reported here are made by running the Python code using PyPy to make the code run faster.

REANNZ. Figure 8(a) shows results for an SDN-enabled IXP that supported the REANNZ research and education network [19]. This real-world policy has 460 OpenFlow 1.0 rules matching on multiple packet headers like `inport`, `dst_ip`, `eth_type`, `src_mac`, etc. Most dependency chains have depth 1 (some lightweight rules have complex dependencies as shown in Figure 3(b)). We replayed a two-day traffic trace from the IXP, and updated the cache every two minutes and measured the cache-hit rate over the two-day period. Because of the many shallow dependencies, all three algorithms have the same performance. The mixed-set algorithm sees a cache hit rate of 84% with a hardware cache of just 2% of the rules; with just 10% of the rules, the cache hit rate increases to as much as 97%.

Stanford Backbone. Figure 8(b) shows results for a real-world Cisco router configuration on a Stanford backbone router [25]. which we transformed into an OpenFlow policy. The policy has 180K OpenFlow 1.0 rules that match on the destination IP address, with dependency chains varying in depth from 1 to 8. We generated a packet trace matching the routing policy by assigning traffic volume to each rule drawn from a Zipf [11] distribution. The resulting packet trace had around 30 million packets randomly shuffled over 15 minutes. The mixed-set algorithm does the best among all three and dependent-set does the worst because there is a mixture of shallow and deep dependencies. While there are differences in the cache-hit rate, all three algorithms achieve at least 88% hit rate at the total capacity of 2000 rules (which is just 1.1% of the total rule table). Note that CacheFlow was able to react effectively to changes in the traffic distribution for such a large number of rules (180K in total) and the software switch was also able to process all the cache misses at line rate. Note that installing the same number of rules in the TCAM of a hardware switch, assuming that TCAMs are 80 times more expensive than DRAMs, requires one to spend 14 times more money on the memory unit.

CAIDA. The third experiment was done using the publicly available CAIDA packet trace taken from the Equinix datacenter in Chicago [26]. The packet trace had a total of 610 million packets sent over 30 minutes. Since CAIDA does not publish the policy used to process these packets, we built a policy by extracting forwarding rules based on the destination IP addresses of the packets in the trace. We obtained around 14000 `/20` IP destination based forwarding rules. This was then *sequentially composed* [18] with an access-control policy that matches on fields other than just the destination IP address. The ACL was a chain of 5 rules that match on the source IP, the destination TCP port and inport of the packets which introduce a dependency chain of depth 5 for each destination IP prefix. This composition resulted in a total of 70K OpenFlow rules that match on multiple header fields. This experiment is meant to show the dependencies that arise from matching on various fields of a packet and also the explosion of dependencies that may arise out of more sophisticated policies. Figure 8(c) shows the cache-hit percentage under various TCAM rule capacity restrictions. The mixed-set and cover-set algorithms have similar cache-hit rates and do much better than the dependent-set algorithm consistently because they splice every single dependency chain in the policy. For any given TCAM size, mixed-set seems to have at least 9% lead on the cache-hit rate. While mixed-set and cover-set have a hit rate of around 94% at the full capacity of 2000 rules (which is just 3% of the total rule table), all three algorithms achieve at least an 85% cache-hit rate.

Latency overhead. Figure 9 shows the latency incurred on a cache-hit versus a cache-miss. The latency was measured by attaching two extra hosts to the switch while the previously CAIDA packet trace was being run. Extra rules initialized with heavy volume were added to the policy to process the ping packets in the TCAM. The average round-trip latency when the ping packets were cache-hits in both directions was $0.71ms$ while the latency for 1-way cache miss was $0.81ms$. Thus, the cost of a 1-way cache miss was $100\mu s$; for comparison, a hardware switch adds $25\mu s$ [27] to the 1-way latency of the packets. If an application cannot accept



(a) REANNZ IXP switch (b) Stanford backbone router (c) CAIDA packet trace

Figure 8: Cache-hit rate vs. TCAM size for three algorithms and three policies (with x-axis on log scale)

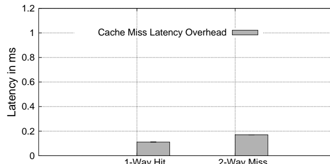


Figure 9: Cache-Miss Latency Overhead

the additional cost of going to the software switch, it can request the CacheMaster to install its rules in the fast path. The CacheMaster can do this by assigning "infinite" weight to these rules.

6.2 Incremental Algorithms

In order to measure the effectiveness of the incremental update algorithms, we conducted two experiments designed to evaluate (i) the algorithms to incrementally update the dependency graph on insertion or deletion of rules and (ii) algorithms to incrementally update the TCAM when traffic distribution shifts over time.

Figure 10(a) shows the time taken to insert/delete rules incrementally on top of the Stanford routing policy of 180K rules. While an incremental insert takes about 15 milliseconds on average to update the dependency graph, an incremental delete takes around 3.7 milliseconds on average. As the linear graphs show, at least for about a few thousand inserts and deletes, the amount of time taken is strictly proportional to the number of flowmods. Also, an incremental delete is about 4 times faster on average owing to the very local set of dependency changes that occur on deletion of a rule while an insert has to explore a lot more branches starting with the root to find the correct position to insert the rule. We also measured the time taken to statically build the graph on a rule insertion which took around 16 minutes for 180K rules. Thus, the incremental versions for updating the dependency graph are ∼60000 times faster than the static version.

In order to measure the advantage of using the incremental TCAM update algorithms, we measured the cache-hit rate for mixed-set algorithm using the two options for updating the TCAM. Figure 10(b) shows that the cache-hit rate for the incremental algorithm is substantially higher as the TCAM size grows towards 2000 rules. For 2000 rules in the TCAM, while the incremental update achieves 93% cache-hit rate, the nuclear update achieves only 53% cache-hit rate. As expected, the nuclear update mechanism sees diminishing returns beyond 1000 rules because of the high rule installation time required to install more than 1000 rules as shown earlier in Figure 7.

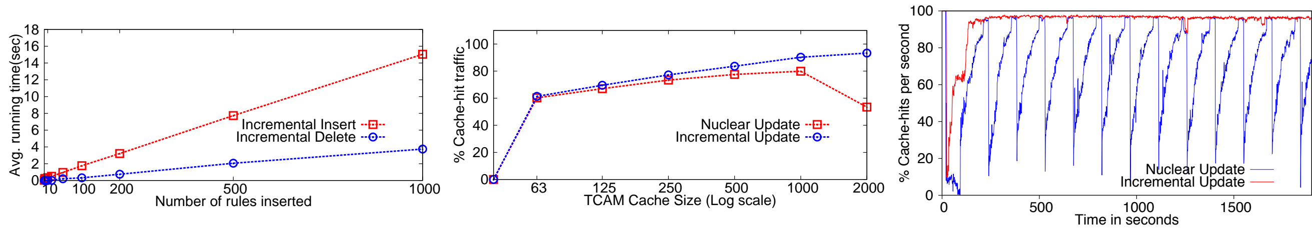
Figure 10(c) shows how the cache-hit rate is affected by the naive version of doing a nuclear update on the TCAM whenever CacheFlow decides to update the cache. The figure shows the number of cache misses seen over time when the CAIDA packet trace is replayed at 330k packets per second. The incremental update algorithm stabilizes quite quickly and achieves a cache-hit rate of 95% in about 3 minutes. However, the nuclear update version that deletes all the old rules and inserts the new cache periodically suffers a lot of cache-misses while it is updating the TCAM. While the cache-hits go up to 90% once the new cache is fully installed, the hit rate goes down to near 0% every time the rules are deleted and it takes around 2 minutes to get back to the high cache-hit rate. This instability in the cache-miss rate makes the nuclear installation a bad option for updating the TCAM.

7. RELATED WORK

While route caching is discussed widely in the context of IP destination prefix forwarding, SDN introduces new constraints on rule caching. We divide the route caching literature into three wide areas: (i) IP route Caching (ii) TCAM optimization, and (iii) SDN rule caching.

IP Route Caching. Earlier work on traditional IP route caching [11–14, 28] talks about storing only a small number of IP prefixes in the switch line cards and storing the rest in inexpensive slow memory. Most of them exploit the fact that IP traffic exhibits both temporal and spatial locality to implement route caching. For example, Sarrar et.al [11] show that packets hitting IP routes collected at an ISP follow a Zipf distribution resulting in effective caching of small number of heavy hitter routes. However, most of them do not deal with cross-rule dependencies and none of them deal with complex multidimensional packet-classification. For example, Liu et.al [28] talk about efficient FIB caching while handling the problem of *cache-hiding* for IP prefixes. However, their solution cannot handle multiple header fields or wildcards and does not have the notion of packet counters associated with rules. Our paper, on the other hand, deals with the analogue of the cache-hiding problem for more general and complex packet-classification patterns and also preserves packet counters associated with these rules.

TCAM Rule Optimization. The TCAM Razor [29–31] line of work compresses multi-dimensional packet-classification rules to minimal TCAM rules using decision trees and multi-dimensional topological transformation. Dong et. al. [32] propose a caching technique for ternary



(a) Incremental DAG update

(b) Incremental/nuclear cache-hit rate

(c) Incremental vs. nuclear stability

Figure 10: Performance of Incremental Algorithms for DAG and TCAM update

rules by constructing compressed rules for evolving flows. Their solution requires special hardware and does not preserve counters. In general, these techniques that use compression to reduce TCAM space also suffer from not being able to make incremental changes quickly to their datastructures.

DAG for TCAM Rule Updates. The idea of using DAGs for representing TCAM rule dependencies is discussed in the literature in the context of efficient TCAM rule updates [33, 34]. In particular, their aim was to optimize the time taken to install a TCAM rule by minimizing the number of existing entries that need to be reshuffled to make way for a new rule. They do so by building a DAG that captures how different rules are placed in different TCAM banks for reducing the update churn. However, the resulting DAG is not suitable for caching purposes as it is difficult to answer the question we ask: if a rule is to be cached, which other rules should go along with it? Our DAG data structure on the other hand is constructed in such a way that given any rule, the corresponding cover set to be cached can be inferred easily. This also leads to novel incremental algorithms that keep track of additional metadata for each edge in the DAG, which is absent in existing work.

SDN Rule Caching. There is some recent work on dealing with limited switch rule space in the SDN community. DIFANE [23] advocates caching of ternary rules, but uses more TCAM to handle cache misses—leading to a TCAM-hungry solution. Other work [35–37] shows how to distribute rules over multiple switches along a path, but cannot handle rule sets larger than the aggregate table size. Devoflow [38] introduces the idea of rule "cloning" to reduce the volume of traffic processed by the TCAM, by having each match in the TCAM trigger the creation of an exact-match rules (in SRAM) the handle the remaining packets of that microflow. However, Devoflow does not address the limitations on the total size of the TCAM. Lu et.al. [39] use the switch CPU as a traffic co-processing unit where the ASIC is used as a cache but they only handle microflow rules and hence do not handle complex dependencies. The Open vSwitch [7] caches "megaflows" (derived from wildcard rules) to avoid the slow lookup time in the user space classifier. However, their technique does not assume high-throughput wildcard lookup in the fast path and hence cannot be used directly for optimal caching in TCAMs.

8. CONCLUSION

In this paper, we define a hardware-software hybrid switch design called CacheFlow that relies on rule caching to provide large rule tables at low cost. Unlike traditional caching solutions, we neither cache individual rules (to respect rule dependencies) nor compress rules (to preserve the per-rule traffic counts). Instead we "splice" long dependency chains to cache smaller groups of rules while preserving the semantics of the network policy. Our design satisfies four core criteria: (1) *elasticity* (combining the best of hardware and software switches), (2) *transparency* (faithfully supporting native OpenFlow semantics, including traffic counters), (3) *fine-grained* rule caching (placing popular rules in the TCAM, despite dependencies on less-popular rules), and (4) *adaptability* (to enable incremental changes to the rule caching as the policy changes).

Acknowledgments. The authors wish to thank the SOSR reviewers for their valuable feedback. We would like to thank Josh Bailey for giving us access to the REANZZ OpenFlow policy and for being part of helpful discussions related to the system implementation. We would like to thank Sepher Assadi for helpful discussions about the computational complexity of the dependency-caching problem. This work was supported in part by the NSF under the grant CNS-1111520, the ONR under award N00014-12-1-0757 and a gift from Cisco.

9. REFERENCES

[1] "TCAMs and OpenFlow: What every SDN practitioner must know." See http://tinyurl.com/kjy99uw, 2012.

[2] B. Stephens, A. Cox, W. Felter, C. Dixon, and J. Carter, "PAST: Scalable Ethernet for data centers," in *ACM SIGCOMM CoNext*, 2012.

[3] "SDN system performance." See http://www.pica8.com/pica8-deep-dive/sdn-system-performance/, 2012.

[4] E. Spitznagel, D. Taylor, and J. Turner, "Packet classification using extended TCAMs," in *ICNP, 2003*.

[5] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, "Dynamic scheduling of network updates," in *SIGCOMM*, 2014.

[6] D. Y. Huang, K. Yocum, and A. C. Snoeren, "High-fidelity switch models for software-defined network emulation," in *HotSDN*, Aug. 2013.

[7] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The design and implementation of Open vSwitch," in *NSDI*, 2015.

[8] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and

S. Ratnasamy, "RouteBricks: Exploiting parallelism to scale software routers," in *SOSP*, 2009.

[9] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: A GPU-accelerated software router," in *SIGCOMM 2010*.

[10] "Intel DPDK overview." See http://tinyurl.com/cepawzo.

[11] N. Sarrar, S. Uhlig, A. Feldmann, R. Sherwood, and X. Huang, "Leveraging Zipf's law for traffic offloading," *SIGCOMM Comput. Commun. Rev. 2012*.

[12] C. Kim, M. Caesar, A. Gerber, and J. Rexford, "Revisiting route caching: The world should be flat," in *Passive and Active Measurement*, 2009.

[13] D. Feldmeier, "Improving gateway performance with a routing-table cache," in *IEEE INFOCOM*, 1988.

[14] H. Liu, "Routing prefix caching in network processor design," in *ICCN 2001*.

[15] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *Proceedings of ICFP '11*.

[16] X. Jin, J. Gossels, J. Rexford, and D. Walker, "Covisor: A compositional hypervisor for software-defined networks," in *NSDI*, 2015.

[17] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Infinite Cacheflow in software-defined networks," in *HotSDN Workshop*, 2014.

[18] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software defined networks," in *NSDI*, 2013.

[19] "REANZZ." http://reannz.co.nz/.

[20] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," in *ACM SIGCOMM*, 2013.

[21] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *NSDI*, 2012.

[22] S. Khuller, A. Moss, and J. S. Naor, "The budgeted maximum coverage problem," *Inf. Process. Lett.*, Apr. 1999.

[23] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with DIFANE," in *ACM SIGCOMM*, 2010.

[24] S. Ghorbani, C. Schlesinger, M. Monaco, E. Keller, M. Caesar, J. Rexford, and D. Walker, "Transparent, live migration of a software-defined network," SOCC '14, (New York, NY, USA), pp. 3:1–3:14, ACM.

[25] "Stanford backbone router forwarding configuration." http://tinyurl.com/oaezlha.

[26] "The CAIDA anonymized Internet traces 2014 dataset." http://www.caida.org/data/passive/passive_2014_dataset.xml.

[27] R. Mittal, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, D. Zats, *et al.*, "Timely: Rtt-based congestion control for the datacenter," in *SIGCOMM*, pp. 537–550, ACM, 2015.

[28] Y. Liu, S. O. Amin, and L. Wang, "Efficient FIB caching using minimal non-overlapping prefixes," *SIGCOMM Comput. Commun. Rev.*, Jan. 2013.

[29] A. X. Liu, C. R. Meiners, and E. Torng, "TCAM Razor: A systematic approach towards minimizing packet classifiers in TCAMs," *IEEE/ACM Trans. Netw*, Apr. 2010.

[30] C. R. Meiners, A. X. Liu, and E. Torng, "Topological transformation approaches to tcam-based packet classification," vol. 19, Feb. 2011.

[31] C. R. Meiners, A. X. Liu, and E. Torng, "Bit weaving: A non-prefix approach to compressing packet classifiers in TCAMs," *IEEE/ACM Trans. Netw.*, vol. 20, Apr. 2012.

[32] Q. Dong, S. Banerjee, J. Wang, and D. Agrawal, "Wire speed packet classification without TCAMs: A few more registers (and a bit of logic) are enough," in *ACM SIGMETRICS*, 2007.

[33] B. Vamanan and T. N. Vijaykumar, "Treecam: Decoupling updates and lookups in packet classification," CoNEXT '11, (New York, NY, USA), pp. 27:1–27:12, ACM.

[34] H. Song and J. Turner, "Nxg05-2: Fast filter updates for packet classification using tcam," in *GLOBECOM'06. IEEE*, pp. 1–5.

[35] Y. Kanizo, D. Hay, and I. Keslassy, "Palette: Distributing tables in software-defined networks," in *IEEE INFOCOM Mini-conference*, Apr. 2013.

[36] M. Moshref, M. Yu, A. Sharma, and R. Govindan, "Scalable rule management for data centers," in *NSDI 2013*.

[37] N. Kang, Z. Liu, J. Rexford, and D. Walker, "Optimizing the 'one big switch' abstraction in Software Defined Networks," in *ACM SIGCOMM CoNext*, Dec. 2013.

[38] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: Scaling flow management for high-performance networks," in *ACM SIGCOMM*, 2011.

[39] G. Lu, R. Miao, Y. Xiong, and C. Guo, "Using CPU as a traffic co-processing unit in commodity switches," in *HotSDN Workshop*, 2012.