Rule-Caching Algorithms 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 ({ecynics}@gmail.com)

ABSTRACT

Software-Defined Networking (SDN) allows control applications to install fine-grained forwarding policies in the underlying switches, using a standard API like OpenFlow. High-speed Ternary Content Addressable Memory (TCAM) allows hardware switches to store these rules and perform a parallel lookup to quickly identify the highest-priority match for each packet. While TCAM enables fast lookups 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.

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 [1]. These rules can match on a wide variety of packet-header fields, and perform simple actions 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. And, combining multiple policies together—say, server load-balancing and routing—that match on different header fields quickly leads to a combinatorial explosion in the number of rules per switch.

In modern hardware switches, these rules are stored in Ternary Content Addressable Memory (TCAM) [2]. 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 [3]. Undoubtedly, emerging switches will support larger rule tables [4, 5], but TCAMs still introduce a fundamental trade-off between rule-table size and other concerns like cost and power. TCAMs introduce around 400 times greater cost [6] and 100 times greater power consumption [7], 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 [8, 9], 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 [10–12] 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 around ten times faster than hardware switches [9]. 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 microflow [13]. So, while software switches are clearly useful at the network edge—particularly in data centers where these switches can run on the end-host servers—they cannot match the "horsepower" of hardware switches that provide hundreds of Gbps of packet processing (and high port density) in the rest of the network.

Fortunately, traffic tends to follow a Zipf distribution, where the vast majority of traffic matches a relatively small fraction of the rules. 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 processor could easily handle traffic with

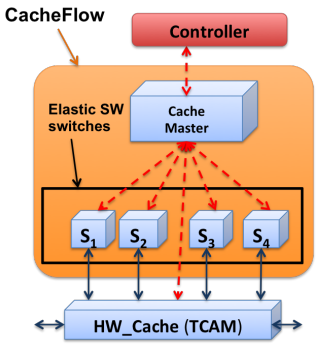


Figure 1: CacheFlow architecture

a 5% "miss rate" in the TCAM. In addition, most rule-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 must preserve the semantics of the OpenFlow interface, including the ability to update rules, query counters, and receive events. CacheMaster also 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. While earlier work on IP route caching [14–17] 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) to generate many more rules. Swapping entire groups of dependent rules in and out of the TCAM would be inefficient, especially if most rules match relatively few packets.

To handle rule dependencies, we construct a compact representation of a prioritized list of rules as a directed acyclic graph (DAG), and design incremental algorithms for adding and removing rules. 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 list of rules, as well as changes in their relative popularity, over time. Experiments with our CacheFlow prototype demonstrate the effectiveness of our algorithms under realistic workloads.

In this paper, we make the following key technical contributions:

- **Incremental rule-dependency analysis:** We develop an algorithm for incrementally analyzing and maintaining rule dependencies—a necessary component of any sound rule-caching scheme.

- **Novel cache-replacement strategies:** We develop algorithms that (i) cache groups of dependent rules, (ii) "splice" dependency chains to cache smaller sets of popular rules, and (iii) a hybrid algorithm combining the best of the two approaches.

- **Implementation and evaluation:** We discuss how CacheFlow preserves the semantics of the OpenFlow interface and shards cache-miss traffic over multiple software switches. Our experiments using Pica8 switches and both synthetic and real workloads show that rule-caching can help process 90% of the total traffic by caching less than 5% of the total rules.

A preliminary version of this work appeared in a workshop paper [18] which briefly discussed ideas about rule dependencies and caching algorithms. In this paper, we develop novel algorithms that help efficiently deal with real deployment constraints like incremental updates to policies and high TCAM update times. We also show that our system can implement large policies on an actual hardware switch as opposed to simulated evaluation done using much smaller policies in the workshop version.

2. IDENTIFYING RULE DEPENDENCIES

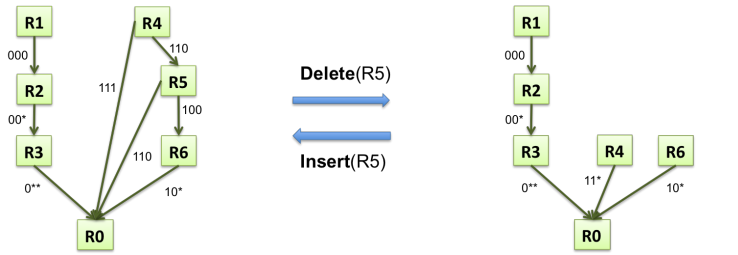
In this section, we show how rule dependencies affect the correctness of any rule-caching technique. We show how to represent all cross-rule dependencies as a graph, and present an efficient algorithm for computing the graph. We also describe how to update the dependency graph incrementally as rules change.

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

Rule	Match	Action	Priority	Weight
R1	000	Fwd 1	6	10
R2	00*	Fwd 2	5	60
R3	0**	Fwd 3	4	30
R4	11*	Fwd 4	3	5
R5	1*0	Fwd 5	2	20
R6	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)

arrives, the switch identifies the highest-priority matching rules and 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 R2 is cached in the TCAM, R1 should also be cached to preserve the semantics of the policy, similarly with R5 and R4.

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

Such a peculiar case does not arise in traditional destination prefix forwarding because a prefix is dependent only on all the prefixes that are strict subsets of itself and 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. For a more concrete example for partial overlaps, one can interpret the first two bits of rules R_4, R_5, and R_6 to match the destination IP prefix, and the last bit to match the destination port field.

2.2 Constructing the Dependency DAG

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);
```

A concise way to capture all of the dependencies 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, all of 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.

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. This dependency defines a parent-child relationship where the rule R is the child and the rule R_i is the parent in the dependency relationship. The arrow in the dependency edge points from the child to the parent. In line 7, the dependency edge also stores the packet space that actually reaches the parent R_i. In line 8, before moving to 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 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. Once such a dependency graph is constructed, all the descendants of a rule in the graph will together constitute the dependencies that a rule carries with it. In other words, 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 in order to maintain correct policy semantics.

2.3 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 simply 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. Before R_5 is inserted, the rules R_4 and R_6 are independent; once R_5 is inserted, R_6 has an indirect dependency on R_4. This is because while packets from R_4 were not impacted by R_6 earlier, now some of them hit R_5 which in turn has its own packets hitting R_6. Also, both R_4 and R_5 have some packets still reaching R_0 even after they hit their immediate parents (R_5 and R_6 respectively). This means that an already-existing edge between R_4 and R_0 has changed as well.

[1]symbolic intersection and subtraction of packets can be done using existing techniques [19]

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 = [R0];
19     FindAffectedEdges(R0, newNode);
20     ProcessAffectedEdges(affEdges);
21     addParents(newNode, potentialParents);
```

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.

2.3.1 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. As we traverse the tree recursively, we update existing edges whose packets intersect with the new rule. We also collect the children and the parents of the new rule (the test is to simply check non-empty intersection of rule predicates along with priorities) that have a direct dependency with the new rule.

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.

(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].

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

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
              // Find if deletion adds an edge
                 E(c, p)
5             if reaches(c, oldRule) ∩ p.match != ∅
              then
6                 add p to potentialParents
7         addParents(C, potentialParents)
8     Remove all edges involving oldRule
```

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).

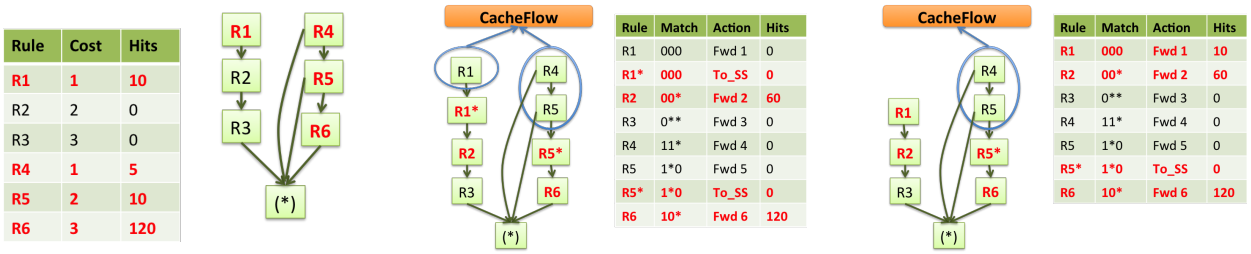
An important observation is that opposed to incremental insert (where we recursively traverse the DAG beginning with R_0), incremental deletion of a rule can be done local to the rule being removed. This is because all three sets of updates involve only the children or parents of the removed rule. For 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.

Figure 2(c) shows the result of deleting R_5 from the dependency graph. Algorithm 3 initializes the potential parents of the child R_4 to $\{R_0\}$. Since line 5 evaluates to False in this case (11* does not intersect with 10*), R_5 is not added to potential parents. Thus in line 7, we

[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.

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

just add one edge (R_4,R_0) with a packet space equals to packets (R_4). Then the edges (R_4,R_5), (R_5,R_6) and (R_5,R_0) are removed.

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.

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 k rules to store in the TCAM[4]. 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 prioritized list.

3.1 Dependent-Set: Caching Dependent Rules

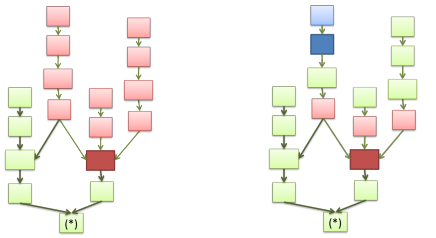
We first present a simple strawman algorithm to build intuition, and then present a new algorithm that avoids caching low-weight rules. Each rule is assigned a "cost" corresponding to the number of rules that must be installed together and a "weight" corresponding to the number of packets expected to hit that rule. For example, R_6 depends on R_4 and R_5, leading to a cost of 3, as shown in Figure 3(a). In this situation, R_2 and R_6 hold the majority of the weight, but cannot be installed simultaneously on the switch, 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. This maximizes total weight, subject to respecting all dependencies. In order to do better, we must restructure the problem.

The current problem of maximizing the total weight can be formulated as a linear integer programming problem, where each rule has a variable indicating whether the rule is installed in the cache. The objective is to maximize the sum of the weights of the installed rules, while installing at most k rules; if rule R_j depends on rule R_i, rule R_j cannot be installed unless R_i is also installed. The problem can be solved with an $O(n^k)$ brute-force algorithm that is expensive for large k. The current problem, however, can also be reduced to an all-neighbors knapsack problem [20], which is a constrained version of the knapsack problem where a node is selected only when all its neighbors are also selected. However, no polynomial time approximation scheme (PTAS) is known for this problem. Hence, we use a heuristic that is modeled on a greedy PTAS for the Budgeted Maximum Coverage problem [21], which is a relaxed version of the all-neighbors knapsack 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.

$$
\begin{aligned}
&\textit{Maximize} && \sum_{i=1}^{n} w_i c_i \\
&\text{subject to} && \sum_{i=1}^{n} c_i \leq k \\
& && c_i - c_j \geq 0 \text{ if } R_i.is\_descendant(R_j) \\
& && \qquad \forall i, j \in \{1, \ldots, n\} \\
& && c_i \in \{0, 1\}\ \forall i \in \{1, \ldots, n\}
\end{aligned}
$$

On the example rule table, this 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. We refer to this

[4]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. In practice, CacheFlow should measure traffic over a time window that is long enough to prevent thrashing, and short enough to adapt to legitimate changes in the workload.



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

Figure 4: Dependent-set vs. cover-set Cost

algorithm as the *dependent-set* algorithm.

3.2 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 3(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`,[5] 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 3(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 3(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.[6] 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 3(a) is 3 as shown in the rule cost table whereas the cost for the new cover set for R_6 in Figure 3(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 4(a) was the entire set of descendants (in light red), but the new cost (in Figure 4(b)) is defined just by the immediate descendants (in light red).

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

[6]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.

Algorithm 4: Mixed-Set Computation

```
1  func Mixed_Set(rule_table) begin
2      create_heap();
3      for r ∈ rule_table do
4          heap.insert(r.dep_set);
5          heap.insert(r.cover_set);
6      total_cost = 0;
7      while total_cost < threshold do
           // pick element with maximum ΔW/ΔC
8          max_rule = heap.get_max();
           // delete both dep-set and cover-set
9          heap.delete(max_rule);
10         if max_rule is dep_set then
11             for r1 ∈ dep_set(max_node) do
12                 heap_delete(r1);
13                 heap_update_ancestors(r1);
14         if max_rule is cover_set then
15             for r1 ∈ cover_set(max_rule) do
16                 heap_update_ancestors(r1);
17         Add_Cache(max_rule.mixed_set);
```

3.3 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 3(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 4(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

Algorithm 5: Incremental TCAM update

```
1  func TCAM_Update(old_cache, new_cache)
   begin
2      for mixed_set ∈ (old_cache − new_cache) do
3          for each rule in mixed_set do
4              if rule is normal then
5                  rule.normal_ref−−;
6                  if rule.normal_ref==0 then
7                      delete_normal(rule)
8              else
9                  rule.star_ref−−;
10                 if rule.star_ref==0 then
11                     delete_star(rule)
12     for mixed_set ∈ (new_cache − old_cache) do
13         for each rule in mixed_set do
14             if rule is normal then
15                 if rule.normal_ref==0 then
16                     install_normal(rule)
17                 rule.normal_ref++;
18             else
19                 if rule.star_ref==0 then
20                     install_star(rule)
21                 rule.star_ref++;
       // Simply retain in TCAM, mixed_set ∈
          new_cache ∩ old_cache.
```

new metric to choose the best candidate rules to cache. We refer to this version as the *mixed-set* algorithm.

Algorithm 4 describes the mixed-set algorithm in detail. For each rule, both dependent-sets and cover-sets are pushed onto a heap (lines 3-5). The heap elements are ordered by the weight to cost ratio of the element's mixed set (The term mixed-set is used to refer to either a dependent-set or a cover-set whichever is appropriate in the context). Once a rule is chosen, both the heap elements associated with the rule are deleted from the heap (line 9). Subsequently, for each rule in the mixed-set chosen, the costs and weights of the ancestors are updated in time for the next iteration (lines 10-16). For example, if rule R_5 is chosen in this iteration, then the entire dependent set (R_4, R_5) is deleted from the heap and the dependent set cost for choosing rule R_6 in the next iteration is changed to just 1 (and not 3).

3.4 Updating the TCAM Incrementally

As the packet traffic distribution of rules change over time, the set of cached rules chosen by our caching algorithms also change over time. This would mean periodically updating the TCAM with a new version of the policy cache.

Updating just the difference will not work.

Simply taking the difference between the two set of cached rules and thereby 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 ($R5*$) and then install the dependent rules (R_5,R_4), there will be a time-period when only the rule R_6 is on 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.

The nuclear option.

An alternative that would work is a nuclear update – simply delete the entire old cache and insert the new cache from scratch. But this is unacceptable because, the rule update time on a TCAM is a non-linear function of the number of rules being inserted or deleted. On a switch that has capacity for 2000 rules in the TCAM, while the first 1000 rules take 5 seconds to be inserted, it takes almost 2 minutes to install the next the 1000 rules. Thus, it is important to minimize the churn in the TCAM when we update the cached rules periodically.

Exploiting composition of mixed sets.

A key property of all the algorithms discussed so far is that each chosen rule along with its mixed(cover/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 **??**(b), the red rule and its cover set can be easily added/removed without disturbing the blue rule and its dependent set. As shown in Algorithm 5, 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 (lines 2-11) and then compose with the new mixed sets (lines 12-21) while retaining the mixed-sets common to both the versions of the cache. Composing two rules to build a cache would simply involve merging their corresponding mixed-sets (and incrementing appropriate reference counters for each rule) and decomposition would involve checking the reference counters before removing a rule from the TCAM. Reference counting is required to take care of overlapping mixed sets. We leave it to the reader to verify that it is indeed safe to leave behind rules on the TCAM that have positive reference counts during the deletion phase[7]. In the example discussed

[7]The intuition is that if a rule has a positive reference count,

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.

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. Then, we explain how CacheMaster preserves the semantics of the OpenFlow interface to the controller.

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 3) 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.

If group tables are not available, CacheMaster can modify the forwarding actions of the cover rules so that, overall, each software switch receives a roughly equal share of cache-miss traffic. It is worth noting that the flexible switch control offered by SDN makes it particularly easy to manipulate the rules cached in the hardware switch to shard the cache-miss traffic.

4.2 Preserving OpenFlow Semantics

To work with unmodified controllers and switches, CacheFlow preserves the 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 [22]. 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[8]. 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 on the tag, pops the tag, and forwards to the designated output port(s).

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

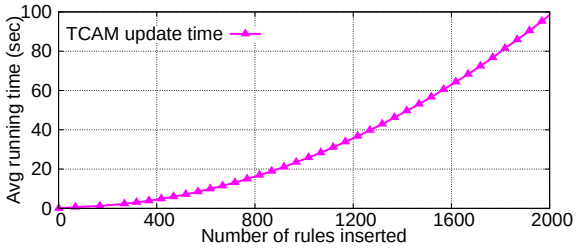


Figure 5: TCAM Update Time

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." Similarly, CacheMaster emulates rule timeouts by installing rules *without* timeouts, and explicitly removing the rules when the software timeout expires, similar to prior work on LIME [23]. 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 sup-

[8]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. Also, CacheMaster does not need to add a tag unless the affected portion of the policy actually differentiates by input port. Finally, 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.

porting 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. Interestingly, the techniques presented in this paper are exactly what the software agent should use to fix this bug!

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 5. 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 do not query the Pica8 switch during rule installation at the start of the experiment. Instead, we wait till the set of installed rules is relatively stable to start querying the counters at regular intervals and relying on the 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 `FlowMod`s 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.

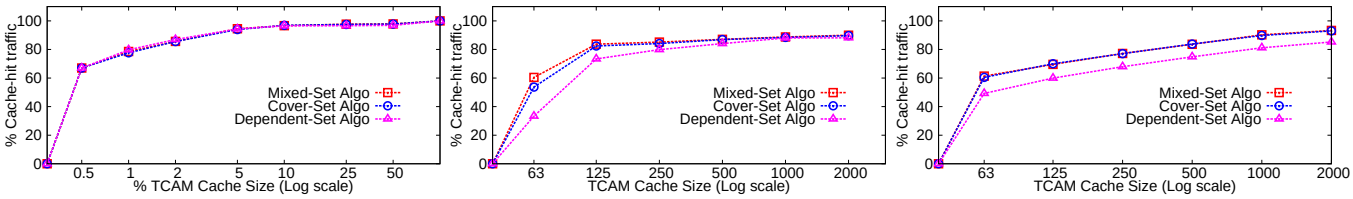
Cache-hit Rate.

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.

Figure 6(a) shows results for an SDN-enabled IXP that supports the REANNZ research and education network [24]. 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. 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 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%.

Figure 6(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 [14] 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). It is worth noting 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.

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* [27] 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



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

Figure 6: Cache-hit rate vs. TCAM size for three algorithms and three policies (with x-axis on log scale)

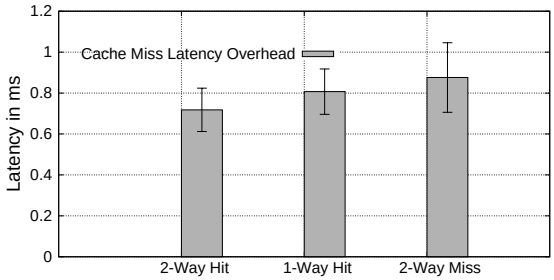


Figure 7: Cache-Miss Latency Measurement

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 6(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.

Figure 7 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 one way cache-miss latency overhead was only $0.10ms$

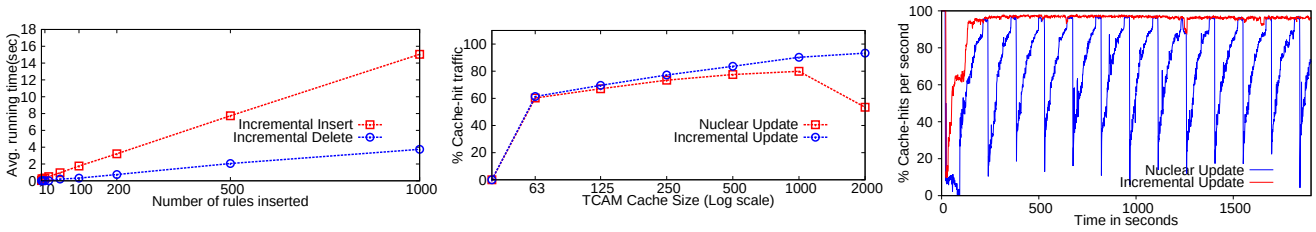
Incremental Algorithms.

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 8(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 8(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 5.

Figure 8(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



(a) Incremental DAG update (b) Incremental/nuclear cache-hit rate (c) Incremental vs. nuclear stability

Figure 8: Performance of Incremental Algorithms for DAG and TCAM update

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 IP route caching [14–17, 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 [14] 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.

There is a long line of research that deals with optimizing packet-classifier rule space in CAMs and TCAMs. 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 rules by constructing compressed rules for evolving flows. Their solution requires special hardware and does not preserve counters. In general, all the above techniques that use compression to reduce TCAM space also suffer from not being to make incremental changes quickly to their data-structures. On the other hand, we rely on splicing long dependencies to reduce rule space using a data structure that is more amenable to incremental changes owing to its compositional nature.

SDN Rule Caching.

There is some recent work on dealing with limited switch rule space for OpenFlow rules in the SDN community. DIFANE [22] advocates caching of ternary rules, but uses more TCAM to handle cache misses—leading to a TCAM-hungry solution. Other work [33–35] shows how to distribute rules over multiple switches along a path, but cannot handle rule sets larger than the aggregate table size. Devoflow [36] 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. [37] 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 preserve rule dependencies.

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).

9. REFERENCES

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM CCR*, vol. 38, no. 2, pp. 69–74, 2008.

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

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

[4] 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.

[5] "Noviflow." `http://noviflow.com/`.

[6] "SDN system performance." See `http://pica8.org/blogs/?p=201`, 2012.

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

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

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

[10] 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.

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

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

[13] "The rise of soft switching." See `http://networkheresy.com/category/open-vswitch/`.

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

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

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

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

[18] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Infinite cacheflow in software-defined networks," in *Proceedings of the HotSDN Workshop*, HotSDN '14, 2014.

[19] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, (Berkeley, CA, USA), USENIX Association, 2012.

[20] G. Borradaile, B. Heeringa, and G. Wilfong, "The knapsack problem with neighbour constraints," *J. of Discrete Algorithms*, vol. 16, pp. 224–235.

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

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

[23] E. Keller, S. Ghorbani, M. Caesar, and J. Rexford, "Live migration of an entire network (and its hosts)," in *HotNets*, Oct. 2012.

[24] "REANZZ." `http://reannz.co.nz/`.

[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] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, "Composing software defined networks," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, USENIX, 2013.

[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," *IEEE/ACM Trans. Netw.*, 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] Y. Kanizo, D. Hay, and I. Keslassy, "Palette: Distributing tables in software-defined networks," in *IEEE Infocom Mini-conference*, Apr. 2013.

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

[35] 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.

[36] 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.

[37] G. Lu, R. Miao, Y. Xiong, and C. Guo, "Using CPU as a traffic co-processing unit in commodity switches," in *HotSDN '12*.