Infinite CacheFlow in 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) enables fine-grained policies for firewalls, load balancers, routers, traffic monitoring, and other functionality. While Ternary Content Addressable Memory (TCAM) enables OpenFlow switches to process packets at high speed based on multiple header fields, today's commodity switches support just thousands to tens of thousands of rules. To realize the potential of SDN on this hardware, we need efficient ways to support the abstraction of a switch with arbitrarily large rule tables. To do so, we define a hardware-software hybrid switch design 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).

Categories and Subject Descriptors

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

Keywords

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

1. INTRODUCTION

Software-Defined Networking (SDN) enables a wide range of applications by applying fine-grained packet-processing rules that match on multiple header fields [1]. For example, an access-control application may match on the "five tuple" (e.g., source and destination IP addresses, transport protocol, and source and destination port numbers), while a server load-balancing application may match on the destination IP address and the source IP prefix. The finer the granularity of the policies, the larger the number of rules in the switches. More sophisticated SDN controller applications combine multiple functions (e.g., routing, access control, monitoring, and server load balancing) into a single set of rules, leading to more rules at even finer granularity.

Hardware switches store these rules in Ternary Content Addressable Memory (TCAM) [2] that performs a parallel lookup on wildcard patterns at line rate. Today's commodity switches support just 2-20K rules [3]. High-end backbone routers handle much larger forwarding tables, but typically match only on destination IP prefix (and optionally a VLAN tag or MPLS label) and are much more expensive. Continued advances in switch design will undoubtedly lead to larger rule tables [4], but the cost and power requirements for TCAMs will continue to limit the granularity of policies SDNs can support. For example, TCAMs are $400X$ more expensive [5] and consume $100X$ more power [6] per Mbit than the RAM-based storage in servers.

On the surface, software switches built on commodity servers are an attractive alternative. Modern software switches process packets at a high rate [7–9] (about 40 Gbps on a quad-core machine) and store large rule tables in main memory (and the L1 and L2 cache). However, software switches have relatively limited port density, and cannot handle multi-dimensional wildcard rules efficiently. While software switches like Open vSwitch cache exact-match rules in the "fast path" in the kernel, the first packet of each microflow undergoes slower user-space processing (i.e., linear search) to identify the highest-priority matching wildcard rule [10].

Instead, we argue that the TCAM in the hardware switch should handle as many of the packets as possible, and divert the remaining traffic to a software switch (or software agent on the hardware switch) for processing. This gives an *unmodified* controller application the illusion of an arbitrarily large rule table, while minimizing the performance penalty for exceeding the TCAM size. For example, an 800 Gbps hardware switch, together with a single 40 Gbps software switch could easily handle traffic with a 5% "miss rate" in the TCAM.

The two main approaches for managing rule-table space are *compression* and *caching*. Rule compression combines

HotSDN'14, August 22, 2014, Chicago, IL, USA.

http://dx.doi.org/10.1145/2620728.2620734.

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

Figure 1: Example Rule Table

rules that perform the same actions and have related patterns [11]. For example, two rules matching destination IP prefixes 1.2.3.0/24 and 1.2.2.0/24 could be combined into a single rule matching 1.2.2.0/23, if both rules forward to the same output port. Unfortunately, we cannot freely combine rules in our setting, because the SDN controller application could query the traffic counters of either of the two original rules at any time. Any viable solution for SDN *must* preserve the semantics expected by controller applications.

As an alternative to rule compression, we treat the TCAM as a *cache* that stores the most popular rules. However, we cannot blindly apply existing cache replacement algorithms, because the rules can have overlapping patterns, leading to complex dependencies between multiple rules. While earlier work on IP route caching [12–15] considered rule dependencies, IP prefixes have simple "containment" relationships rather than complex partial overlaps of rules. The partial overlaps can also lead to long dependency chains. Sophisticated SDN applications that combine multiple functions can easily lead to even longer dependency chains. In this setting, swapping entire groups of dependent rules in and out of the cache would be extremely inefficient, especially if most of these rules match very little traffic.

In this paper, we show how to analyze rule patterns to compute a graph that captures all of the dependencies between the rules in a switch. Then, we introduce a novel "splicing" technique that breaks long dependency chains, allowing us to cache much smaller groups of rules. Splicing involves creating a few new rules that "cover" a large number of unpopular rules in a dependency chain, to avoid polluting the cache. Splicing preserves rule-table space for the more important rules that match a large fraction of the traffic, while still respecting rule dependencies. In addition, our technique extends naturally to handle changes in the list of rules over time. The dependency-graph representation is inherently modular, allowing our algorithm to compute the dependency graph and the "cover" sets incrementally as the rules change. Experiments with our prototype CacheFlow implementation demonstrate the effectiveness of our algorithm under realistic workloads.

We discuss the rule-dependency problem and various rule-caching algorithms in section 2. We discuss the CacheFlow system design, a prototype implementation and a simulated evaluation in section 3. We then briefly discuss related work in section 4 and conclude with section 5.

2. CACHEFLOW ALGORITHM

In this section, we present CacheFlow's algorithm for placing rules in a TCAM with a 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.

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

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

  return deps;
}
```

Figure 2: Building the Dependency Graph

The input to the rule-caching problem is a prioritized list of 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. The output is a prioritized list of k rules to store in the TCAM[1]. 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.

2.1 Computing Rule Dependencies

At regular intervals, the update algorithm decides which rules to cache in the TCAM, based on their weights. The problem is easy to solve if the rules have disjoint patterns (e.g., microflow rules with no wildcard bits). In this case, each rule is independent, and a greedy algorithm could cache the k rules with the highest weights.

However, the greedy algorithm is incorrect when rules have dependencies. Figure 1 shows an example with six rules that match on a ternary format. If the TCAM can store four rules ($k = 4$), we cannot select the four rules with highest weight (i.e., R_3, R_4, R_5, and R_6), because packets that should match R_1 (with pattern 0000) would match R_3 (with pattern 000*); similarly, some packets (with pattern 11**) that should match R_2 would match $R4$ (with pattern 1*1*). That is, rules R_3 and R_4 *depend* on rules R_1 and R_2, respectively.

A dependency exists between any two rules if the matches in the rules intersect (e.g., R_6 is dependent on R_4). When a rule R is cached in the TCAM, the corresponding dependent rules should also move to the TCAM. However, checking for intersecting patterns does not capture all of the rule dependencies. R_6 also depends on R_2 (even though the matches of R_2 and R_6 do not intersect), because the match 11** overlaps with that of R_4 (1*1*). If the TCAM stored only R_4 and R_6, packets that should match R_2 would inadver-

[1]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) Dependent-set Algo. (b) Cover-set Algo. (c) Dependent-set Cost (d) Cover-set Cost

Figure 3: Dependent-set vs. Cover-set Algorithms (L_0 cache rules in red)

tently match R_4. Therefore we need to define carefully what constitutes a dependency to handle such cases properly.

The algorithm in Figure 2 captures all such dependencies. Rather than considering the dependencies for each rule separately, our algorithm constructs a single dependency graph, as shown in Figure 3(a). To find the rules that depend on R, the algorithm scans the rules R_i with lower priority than R in order of decreasing priority. As the algorithm proceeds, it keeps track of the set of packets that can reach each successive rule (the variable `reaches`). For each such new rule, it determines whether the predicate associated with that rule intersects the set of packets that can reach that rule. If it does, there is a dependency. Moreover, the rule R_i will occlude lower-priority rules. Hence, we subtract R_i's predicate from the current `reaches` set.

2.2 Caching Groups of 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_5 depends on R_1 and R_3, leading to a cost of 3, as shown in Figure 3(a). In this situation, R_5 and R_6 hold the majority of the weight, but cannot be installed simultaneously on the switch, as installing either has a cost of 3 and together they do not fit. The best we can do is to install rules R_1, R_2, R_4, 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 [16], which is a constrained version of the knapsack problem where a node is selected only when all its neighbors are also selected [2]. 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 [17], 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.

On the example rule table, this greedy algorithm selects R_6 first (and its dependent-set $\{R_2, R_4\}$), and then $R1$ which brings the total cost to 4. Thus the set of rules in the TCAM are R_1, R_2, R_4, and R_6. We refer to this algorithm as the *dependent-set* algorithm.

2.3 Splicing Long Chains of Dependent Rules

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_4^* with match 1*1* and action `forward_to_SW_switch`,[3] along with the low-priority rule R_6. Similarly, we can create a new rule R_3^* to break dependencies on R_5. We avoid installing higher-priority, low-weight rules like R_2, and instead have the high-weight rules R_5 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_SW_switch` action. For example, the cover-set for rule R_6 is the rule R_4^* in Figure 3(b); sim-

[2]The reduction involves modeling all the dependencies of a node as its neighbors in the knapsack problem

[3]This is just a standard forwarding action out some port on the hardware switch that is connected to a software switch.

ilarly, R_3^* is the cover-set for R_5. The rules defining these `forward_to_SW_switch` actions may also be merged[4], if necessary, to reduce the cost even further. 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_4^* and R_6. To take a more general case, the old cost for the red rule in Figure 3(c) was the entire set of ancestors (in light red), but the new cost (in Figure 3(d)) is defined just by the immediate ancestors (in light red).

Combining the best of the two techniques: 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_4 in Figure 3(b), 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. As shown in Figure 3(d), 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). 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 candidate rules to cache. We refer to this version as the *mixed-set* algorithm, and evaluate its performance in Section 3.2.

2.4 Updating the Rules Incrementally

A key property of all the algorithms discussed so far, is that each chosen rule and its cover/dependent-set can be added/removed almost independently of the rest of the chosen rules. In other words, the mixed-sets for two rules are easily composable and decomposable. Composing two rules to build a cache would simply involve merging the corresponding two sets (and incrementing appropriate reference counters for each rule) and decomposition would involve reference counting before removing a rule from the TCAM. This makes the all the algorithms discussed here amenable to incremental change. For example, in Figure 3(d), the red rule and its cover-set can be easily added/removed without disturbing the blue rule.

We also developed algorithms to incrementally maintain the dependency graph so that when a new rule is inserted or an old rule is removed from the dependency graph, the graph is still maintained without incurring the complexity of the static algorithm shown in Figure 2. But we omit these ideas for lack of space. The key intuition is to store edge-related metadata so that we can incrementally maintain the graph while manipulating only the relevant edges.

3. CACHEFLOW SYSTEM DESIGN

CacheFlow combines the high speed of hardware switches with the large rule tables of software switches, to offer the abstraction of a single, fast switch with arbitrarily large capacity. CacheFlow consists of a CacheMaster module that receives OpenFlow commands from the controller, and uses the OpenFlow protocol to distribute rules to the underlying switches, as shown in Figure 4. Having multiple software switches allows CacheFlow to "shard" the cache-miss traffic over multiple software switches (by assigning different "cover" rules to different output ports), for higher throughput and rule capacity. Packets stay in the "fast path" in the data plane, reducing the performance penalty for a cache miss.

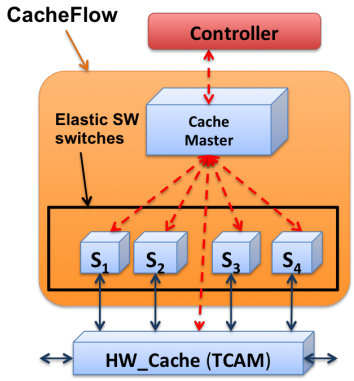

Figure 4: CacheFlow Architecture

3.1 Preserving OpenFlow Semantics

The algorithms in Section 2 preserve the semantics of OpenFlow rule priorities and counters for the cache-hit traffic. CacheFlow also ensures that the software switches handle cache-miss traffic correctly, as well as other aspects of the OpenFlow protocol (e.g., barriers, rule timeouts, etc.) are handled correctly, so our system can work with unmodified controller applications.

Preserving inports/outports: CacheFlow installs three kinds of rules in the hardware switch: (i) fine-grained rules that apply part of the policy (a "hit"), (ii) coarse-grained rules that forward packets to a software switch (a "miss"), and (iii) coarse-grained rules that direct return traffic from the software switches to the right output port(s), similar to the tunneling mechanisms used in DIFANE [18]. 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 the software switches can apply rules that depend on the inport[5]. 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 hard-

[4] 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.

[5]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, CacheFlow 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.

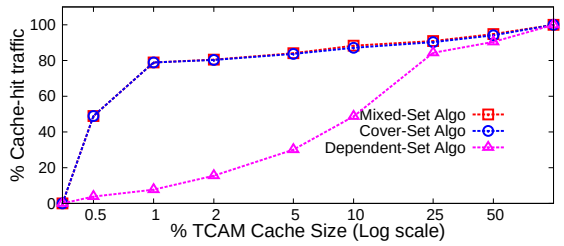

Figure 5: ClassBench Access Control Policy

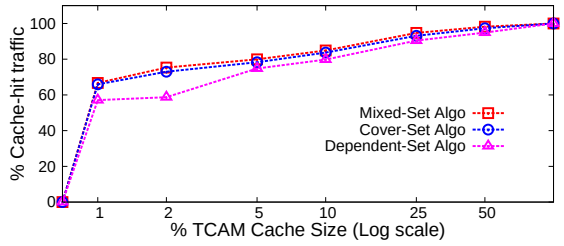

Figure 6: Stanford Backbone Router Policy

ware switch simply matches on the tag, pops the tag, and forwards to the designated output port(s). If a cache-miss rule has an action that sends the packet to the controller, the CacheMaster transforms the `packet_in` message from the software switch by (i) copying the inport from the tag into the inport of the `packet_in` message and (ii) stripping the tag from the packet before sending the message to the controller.

Traffic counts, barrier messages, and rule timeouts: CacheFlow preserves the semantics of OpenFlow 1.0 constructs like queries on traffic statistics, barrier messages, and rule timeouts by emulating all of these features in the CacheMaster—i.e, the behavior of the switches maintained by CacheFlow is no different from that of a single OpenFlow switch with infinite rule space. 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, CacheFlow emulates rule timeouts by installing rules *without* timeouts, and explicitly removing the rules when the software timeout expires, similar to prior work on LIME [19].

3.2 Implementation and Evaluation

We implemented a prototype for CacheFlow in Python on top of the Ryu controller platform. At the moment, the prototype transparently supports the semantics of the OpenFlow 1.0 features mentioned earlier, except rule timeouts and barrier messages. We ran the prototype on a collection of two Open VSwitch 1.10 instances where one switch acts as the hardware cache (where the rule-table capacity is limited by CacheFlow) and another acts as the software switch that holds the cache-miss rules. We evaluate our prototype for three algorithms (dependent-set, cover-set, and mixed-set) and three policies, and measure the cache-hit rate.

The first policy is a synthetic Access Control List (ACL) generated using ClassBench [20]. The policy has 10K rules that match on the source IP address, with long dependency chains with maximum depth of 10. In the absence of a traffic trace, we assume the weight of each rule is proportional to the portion of flow space it matches. Figure 5 shows the cache-hit percentage across a range of TCAM sizes, expressed relative to the size of the policy. The mixed-set and cover-set algorithms have similar cache-hit rates and do much better than the dependent-set algorithm because they splice the longer dependency chains in the policy. While mixed-set and cover-set have a hit rate of around 87% for 1% cache size (of total rule-table), all three algorithms have around 90% hit rate with just 5% of the rules in the TCAM.

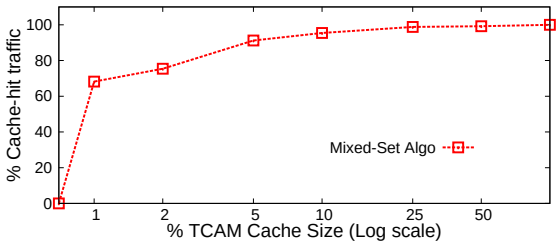

Figure 7: REANNZ IXP Policy

Figure 6 shows results for a real-world Cisco router configuration on a Stanford backbone router [21], which we transformed into an OpenFlow policy. The policy has 5K OpenFlow 1.0 rules that match on the destination IP address, with dependency chains that vary in depth from 1 to 6. We analyzed the cache-hit ratio by assigning traffic volume to each rule proportional to the size of its flow space. 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 70% hit rate with a cache size of 5% of the policy.

Figure 7 shows results for an SDN-enabled Internet eXchange Point (IXP) that supports the REANNZ research and education network [22]. 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 and hence we only show the mixed-set algorithm in the figure. The mixed-set algorithm sees a cache hit rate of 75% 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%.

4. RELATED WORK

Earlier work on IP route caching [12–15] store only a small number of IP prefixes in the switch line cards and 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. However, most of them do not deal with cross-rule dependencies and none of them deal with complex multidimensional packet-classification. The TCAM Razor [11, 23] line of work compresses multi-dimensional packet-classification rules to minimal TCAM rules using decision trees and multi-dimensional topological transformation. Dong et. al. [24] proposes a caching technique for ternary rules by creating new rules out of existing rules

that handle evolving traffic but requires special hardware and does not preserve counters. In general, the above techniques that use compression to reduce TCAM space suffer from not being able to (i) preserve rule counters and (ii) make efficient incremental changes. In recent SDN literature, DIFANE [18] advocates caching of ternary rules, but uses TCAM to handle cache misses—leading to a TCAM-hungry solution. Other work [25,26] shows how to distribute rules over multiple switches along a path, but cannot handle rule sets larger than the aggregate table size. vCRIB [27] redirects packets over a longer path before the entire policy is applied and their partitioning approach is not amenable to incremental change or transparency.

5. CONCLUSION

CacheFlow enables fine-grained policies in SDNs by optimizing the use of the limited rule-table space in hardware switches, while preserving the semantics of OpenFlow. Cache "misses" could be handled in several different ways: (i) an inline CPU or network processor in the data plane of the hardware switch (if available), (ii) one or more software switches (to keep packets in the "fast path", at the expense of introducing new components in the network), (iii) in a software agent on the hardware switch (to minimize the use of link ports and bandwidth, at the expense of imposing extra CPU and I/O load on the switch), or (iv) at the SDN controller (to avoid introducing new components while also enabling new network-wide optimizations, at the expense of extra latency and controller load). In our ongoing work, we plan to explore these trade-offs, and also evaluate our algorithms under a wider range of SDN policies and workloads.

Acknowledgments. The authors wish to thank the HotSDN reviewers and members of the Frenetic project for their feedback. We would especially 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. This work was supported in part by the NSF under the grant TS-1111520; the ONR under award N00014-12-1-0757; and a Google Research Award.

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

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

[7] 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.

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

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

[10] "The rise of soft switching." See `http://tinyurl.com/bjz8469`.

[11] 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.

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

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

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

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

[16] G. Borradaile, B. Heeringa, and G. Wilfong, "The knapsack problem with neighbour constraints," *J. of Discrete Algorithms*, vol. 16, pp. 224–235, Oct. 2012.

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

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

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

[20] D. E. Taylor and J. S. Turner, "Classbench: A packet classification benchmark," in *IEEE INFOCOM*, 2004.

[21] "Stanford backbone router forwarding configuration." `http://tinyurl.com/o8glh5n`.

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

[23] 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.

[24] 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.

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

[26] 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.

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