Infinite CacheFlow in Software-Defined Networks

Naga Katta
Princeton University

Jennifer Rexford
Princeton University

David Walker
Princeton University

Abstract

Software-Defined Networking (SDN) enables firewalls, load balancers, routers, traffic monitoring, and other functionality to be built using commodity hardware. 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 an *infinite switch*. To do so, we define a new hardware-software architecture called CacheFlow. In designing CacheFlow, we observe that any infinite SDN switch must satisfy four core criteria: (1) *elasticity* (combining the best of hardware and software switches), (2) *transparency* (supporting native OpenFlow semantics faithfully), (3) *fine-grained* rule caching (placing popular rules in the TCAM, despite dependencies on less-popular rules), and (4) *malleability* (to enable incremental changes to rule caching as traffic demands change). We propose a new architecture called CacheFlow that, unlike previous caching systems, supports all these properties while rewriting, reordering and caching important switch rules using a novel set of algorithms specifically tailored for the challenges and opportunities in SDN.

1. Introduction

Software-Defined Networking (SDN) enables a wide range of applications by applying fine-grained packet-processing rules in the switches [1]. While a routing application may forward packets based on the destination prefix, more sophisticated applications (e.g., access control and server load balancing) match on multiple packet-header fields. Even basic routing in transit backbones, large data centers, and IPv6 networks would require many rules, and finer-grained policies push the number of rules even higher.

Hardware switches implement these rules using Ternary Content Addressable Memories (TCAM) [2] that perform a parallel lookup on wildcard patterns at high speed. Today's commodity switches support just 2K to 20K rules [3]. High-end backbone routers have much larger forwarding tables, but typically match only on destination IP prefix (and VLAN tag or MPLS label). 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 $100X$ more power-consuming [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 achieve a high rate of packet processing [7–9] (about 39Gbps per port) and store large rule tables in main memory (and the L1 and L2 cache). However, software switches have relatively limited port density, and handling multi-dimensional wildcard rules in software is difficult. 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., a linear search) to identify the highest-priority matching wildcard rule [10].

In this paper, we present the design and implementation of an architecture called CacheFlow, which presents unmodified SDN applications with the illusion of an *infinite* rule table using a new set of rule-caching algorithms tailored for SDNs. While rule caching dates back to early work on IP route caching [11–14] and to some recent work on TCAMs [15, 16], SDNs introduce several new challenges and opportunities:

Elasticity: An infinite switch architecture must use all available resources efficiently. CacheFlow combines the fast processing of hardware switches with the large rule-table space of software. While Cache-Flow logically sits between the controller and the hardware switch, the system could easily run on the controller, a local agent on the hardware switch, or an adjacent software switch. Moreover, flow space may be partitioned so that multiple software switches can handle cache misses for a single hardware switch. The direct control of network switches offered by SDN makes it particularly easy to manipulate rule caches in the hardware switch and flexibly forward cache misses to an appropriate backup switch.

Transparency: SDN applications can add or remove rules, can query traffic counters associated with each rule and may expect the switch to automatically delete rules when a timeout expires. Thus, any rule-manipulation done by a caching abstraction has to be transparent with respect to

these expectations. Thus, CacheFlow cannot combine rules that have related patterns, as is common in many rule compression techniques [17], without losing the traffic counters. In addition, CacheFlow must also maintain traffic statistics for rules not currently in the cache, to return the correct responses to queries. CacheFlow must also approximate hard and soft timeouts.

Fine granularity: SDN applications can generate rules with overlapping patterns in multiple dimensions. For example, a rule matching on source port 80 overlaps with another rule matching on destination IP prefix. Unfortunately, overlapping rules form dependency chains and complicate the process of placing select rules. Unable to break these dependences, past caching systems were forced to install rules in groups, even though many rules handle little traffic. Our new algorithms "splice" these dependency chains to cache much smaller groups of rules. Here again, we take advantage of the rule priority structure in an SDN flowtable – we install additional rules at the appropriate priority to maximally splice dependency chains and direct minimal traffic to a backup software switch.

Malleability: A key benefit of SDN is the ability to adapt rapidly to change. Any caching system must be equally dynamic. When the application makes incremental changes in forwarding policy, or the load shifts and the caching policy must change, an effective SDN cache must react quickly and minimize churn. This demands new "pay-as-you-go" algorithms that can make incremental adjustments to the rule cache, without unnecessarily rearranging other rules. Our algorithm by virtue of the compositional nature of its cache construction, adds(removes) sets of rules to(from) the cache with minimal disruption to the rest of the cache.

Related Work Most of the IP route caching literature [11–14] do not deal with complex packet classification patterns. The closest work related to our research is by Dong et. al. [15] which discusses a caching technique for ternary rules but it requires special hardware and does not preserve counters due to rule compression. Difane [16] uses auxiliary TCAMs as secondary caches to the switches already in the network and hence is a TCAM-hungry solution.

Summary: CacheFlow is a new architecture and a new set of algorithms that implement the abstraction of an infinite switch for software-defined networks. The following sections explain the architectural design (Section 2), the cache-update algorithms (Section 3) and the techniques our prototype uses to preserve transparency (Section 4).

2. CacheFlow

CacheFlow is a transparent caching layer that logically sits between a SDN controller and a collection of OpenFlow switches. Many deployment scenarios are possible, including CacheFlow running on the controller, a CPU on the hardware switch, or servers near the hardware switch.

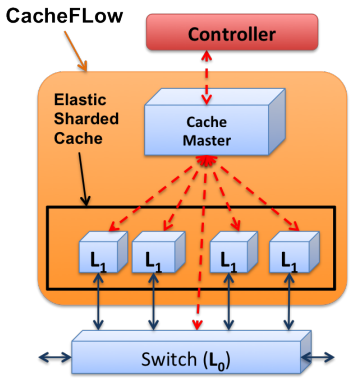

Figure 1: CacheFlow architecture

2.1 CacheFlow Architecture

CacheFlow makes a collection of hardware and software switches act like a single switch with an infinite rule capacity. CacheFlow consists of 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 1. The switches form a cache hierarchy, where packets that experience a cache miss in one layer are forwarded in the data plane to a switch at the next layer for handling. That is, the CacheMaster is a purely control-plane component (with control sessions shown as dashed lines), and the OpenFlow switches forward packets in the data plane (as shown by the solid lines).

Figure 1 shows a simple configuration where one hardware switch connects directly to a *sharded cache* consisting of multiple software switches. On the one hand, the hardware switch provides high port density, high throughput, and a modestly-sized TCAM. On the other hand, the software switches provide high rule capacity at reasonable throughput to handle "cache misses" in the hardware switch. Having multiple software switches allows CacheFlow to "shard" both traffic and rules over multiple switches, to achieve higher bandwidth and rule capacity. Packets stay in the "fast path" in the data plane, reducing the performance penalty for a cache miss.

CacheFlow is designed to be elastic. The cache hierarchy can be *wide* (with multiple switches at each layer) or *tall* (with multiple layers). Having multiple switches in a layer results in a larger rule capacity with higher aggregate throughput. Having multiple layers consumes fewer ports on the hardware switch, while offering high rule capacity for a "long tail" of unpopular rules. The CacheMaster can decide dynamically how many switches to have at each layer, including a mix of hardware and software switches, and how to partition the rules.

2.2 CacheFlow Deployment Options

CacheFlow could run in a variety of locations, with different performance and scalability trade-offs:

On a CPU in the hardware switch: CacheFlow could run on a processor that is part of the hardware switch. In this scenario, CacheFlow would consist of a CacheMaster and a single software switch that handles all cache-miss packets. This has the advantage of low latency on cache misses, and not requiring any additional hardware components; however, cache-miss throughput is limited by the processing and I/O capacity of the local processor.

Near the hardware switch: CacheFlow could run on a collection of servers and switches near the hardware switch. Running on separate components gives CacheFlow more processing and I/O resources, and allows those resources to expand and contract dynamically. Running close to the hardware switch minimizes the bandwidth overhead and latency for handling cache misses.

On the SDN controller: CacheFlow could run on the logically-centralized controller. This has the advantage of not requiring additional hardware components, at the expense of higher latency and overhead on cache misses. Running CacheFlow at the controller also enables network-wide optimizations, such as pushing rules into the multiple switches along a path, in response to shifts in traffic loads.

More generally, CacheFlow could have a hybrid deployment that splits functionality between the SDN controller and components on/near each hardware switch. The logically-centralized portion of Cache-Flow would perform the network-wide optimizations for pushing rules, while the local logic on or near the switches would minimize latency and bandwidth overhead for handling cache misses. Exploring the range of options is an interesting avenue for future research.

3. Cache Update Algorithm

In this section, we present CacheFlow's algorithm for placing rules in a TCAM with a limited space. But the same algorithm can be easily extended to describe a way to recursively place rules in a cache hierarchy.

The input is a prioritized list 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. The cache update algorithm must compute 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.

[1] Note that our problem formulation does not simply install rules on cache misses. 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.

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 2: Example rule table

3.1 Computing Rule Dependencies

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

However, the greedy algorithm is incorrect when rules have dependencies. Figure 2 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*); simiarly, 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.

To solve the above problem, one could define a dependency that exists between any two rules if the matches in the rules intersect. In that case, for a given rule R, the dependent set is given by all the rules that have a non-empty intersection with it. When a rule R is chosen to be cached in the TCAM, the corresponding dependent set is also packed along with the R and sent to the TCAM.

However, this simple definition does not capture all the rule dependencies in a ternary rule table. According to the previous definition of a dependency, only the match 10*1 in rule R_6 overlaps with the match 1*1* in R_4, making R_6 dependent only on R_4. However, 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 inadvertently match R_4. Therefore one needs to carefully define what constitutes a dependency so that such cases are handled properly.

The algorithm in Figure 3 captures all such dependencies properly. Rather than considering the dependencies for each rule separately, our algorithm constructs a single dependency graph, as shown in Figure 4(a). To find the rules that depend on R, the algorithm scans the rules R_i with lower priority

```
// 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 3: Building the dependency graph

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.

3.2 Caching Under Rule Dependencies

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 2. 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, however, 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 integer programming problem can be solved with an $O(n^k)$ brute-force algorithm but is computationally expensive for large k. The current problem, however, can also be reduced to a Budgeted maximum coverage problem [18], which has an efficient greedy approximation algorithm. At each stage, this greedy algorithm chooses a set of rules that maximizes the ratio of combined rule weight to combined rule cost, until the total cost reaches k. This algorithm achieves an approximation ratio of $1 - \frac{1}{e}$.

If one follows this greedy algorithm for the example rule-table, one selects R_6 first (and also the dependent set $\{R_2, R_4\}$ along with it) and then $R1$ which brings the total cost to 4. Thus the set of rules that will now be in the TCAM are R_1, R_2, R_4 and R_6. We refer to this algorithm as the *dependent-set* algorithm.

3.3 Avoiding Caching Low-Weight Rules

Respecting rule dependencies can lead to high costs, especially if a high-weight rule depends on a large number of 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 allowing the set of rules to be modified in various semantics-preserving ways instead of simply attempting to pack the best set of existing rules in to the available cache space—this is the key observation that leads to our superior algorithms. In particular, our algorithm truncates the dependency chain by creating a small number of new rules that cover many low-weight rules and send packets to CacheFlow.

For the example in Figure 4(a), instead of selecting all dependent rules for R_6, we calculate new rules that cover the set of packets that would otherwise incorrectly hit R_6. The extra rules direct these packets to CacheFlow, thereby truncating the dependency chain. For example, we can install a high-priority rule R_4^* with match 1*1* and action `forward_to_L1_cache`,[2] along with the low-priority rule R_6. Similarly, we can create new rule R_3^* to trucate dependencies on R_5. Extending the policy with these new rules allows the two high-weight rules R_5 and R_6 to inhabit the cache simultaneously, as shown in Figure 4(b). By truncating dependencies, we avoid installing higher-priority, low-weight rules like R_2 and can make significantly better caching decisions.

In general, to carry out the algorithm, we 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_L1_cache` action. For example, the cover set for rule R_6 is the rule R_4^* in Figure 4(b); similarly, R_3^* is the cover set for R_5. The rules defining these `forward_to_L1_cache` actions may also be merged, if necessary.[3] 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

[2] This is just a standard forwarding action out some port connected to the cache.

[3] To preserve OpenFlow semantics pertaining to hardware packet counters, policy rules cannot be compressed. However, intermediary rules used for forwarding cache misses to CacheFlow may be compressed, while still tracking packet counters accurately.

(a) Dependent Set Algo. (b) Cover Set Algo. (c) Dep. Set Cost (d) Cover Set Cost

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

long chains of dependencies. For example, the old dependent set cost for the rule R_6 in Figure 4(a) is 3 as shown in the rule cost table whereas the cost for the new cover set for R_6 in Figure 4(b) is only 2 since we only need to cache R_4^* and R_6. To take a more general case, the old cost for the red rule in Figure 4(c) was the entire set of ancestors (in light red), but the new cost (in Figure 4(d)) is defined just by the immediate ancestors (in light red).

In this setting with cover sets, we can apply the same greedy covering algorithm with the new cost metrics, and achieve a higher hit rate in the cache. We refer to this version as the *cover-set* algorithm.

To see how this algorithm performs against the dependent-set algorithm, We evaluated both of them against a synthetic firewall policy with 100K rules. The policy consists of access control chains of depths ranging from 5 and 15. The rules are randomly assigned traffic volume according to a zipf distribution. Figure 5 shows the amount of traffic that "hits" the TCAM (L_0) cache as we vary the TCAM rule-space (k). The cover-set algorithm consistently outperforms the dependent-set algorithm by holding around 18% more traffic in the dataplane while the cache size is increased from 1% to 8% of the total number of firewall rules. The cover-set algorithm achieves an 80% cache-hit rate with just 8000 TCAM rules.

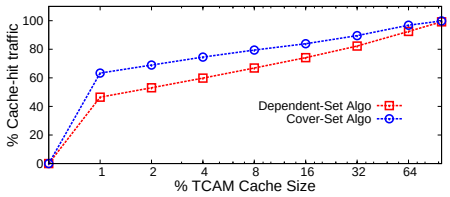

Figure 5: Comparing % Cache hit Traffic

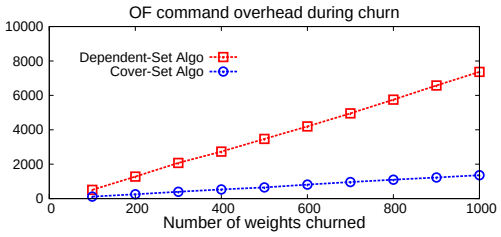

Figure 6: Comparing OpenFlow overhead

3.4 Malleable Caching

A key property of the new algorithm for finding the optimal set of cached rules is that each chosen rule and its cover set can be added/removed independently of the rest of the chosen rules. In other words, the set covers for two rules are easily composable and decomposable. This makes the cover-set algorithm more *malleable* than the dependent-set algorithm. For example, it is easy to incrementally run the cover-set algorithm to find the new set of cached rules, when the traffic weights change. But in the dependent-set algorithm, in order to add or remove a rule R and its dependencies from the TCAM, we have to be careful not to break the dependent sets of other rules, which may have significant overlap. For example, in Figure 4(d), the red rule and its cover set can be easily added/removed without disturbing the blue rule and its dependencies whereas you need to do additional analysis for such a composition to be possible in Figure 4(c).

Even after finding the exact changes to the TCAM rule-table, the dependent-set algorithm may have to evict or add entire chains all at once, whereas the cover-set algorithm can update the TCAM with a small number of OpenFlow rules. This can help avoid a large transient increase in cache-misses during the update. It also decreases the control-plane bandwidth and the TCAM policy update time. Continuing with the experimental setup described before, Figure 6 shows the number of OpenFlow rules that are affected when there is churn in the traffic weights. Given an L_0 TCAM of size $8K$ rules, the x-axis records the number m of rules that have their weights changed. The y-axis records the number of rules (including dependent rules) that must be replaced due to the change in weights. The dependent-set algorithm touches approximately $5\times$ more rules than the cover-set algorithm.

3.5 Handling Low Weight Traffic

The above algorithm computes the set of rules to cache in the hardware switch. It also has to distribute traffic hitting the uncached rules equally to the set of lateral L_1 software switches so that the cache misses are handled together by them effectively. To do this, a post-processing stage scans the final rule-cache (the red rules shown in Figure 4(b)), and replaces each of the "Fwd_to_L_1" actions with an appropriate "Fwd_to_L_1^i" action where L_1^i represents the ith L_1 software switch. This is done in such a manner that the cumulative weight of traffic that is directed to these software switches is equally balanced. The key insight is that traffic matching every uncached rule hits a cover set rule of the form R_j^*. Thus we assign the weight of an R_j^* rule to be equal to the uncached traffic hitting it. Then we assign the complete set of R_j^* rules to the set of L_i software switches in such a manner that the total weight of R_j^* rules is balanced across these switches.

4. Preserving OpenFlow

Here, we give a brief overview of how to ensure transparency when handling the OpenFlow messages.

(Un)installing rules: The CacheMaster receives the entire set of `FlowMod` messages from the controller, stores copies locally, runs the cache update algorithm, and distributes rules to the underlying switches in the cache hierarchy. CacheMaster adds default rules on each switch to handle cache misses. The CacheMaster 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 handle return traffic from the software switches, similar to mechanisms used in DIFANE [16]. The hardware switch tags cache-miss packets with the input port (e.g., using VLAN tags) so the software switches can apply rules that depend on that information. 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 chosen output port.

Rule timeouts: The controller may install rules with hard timeouts (that expire after a fixed time) or soft timeouts (that expire after a period of inactivity). However, in CacheFlow, having the switches automatically delete rules could interfere with the handling of rule dependencies. Instead, CacheMaster installs rules with no timeouts, and simulates the timeouts itself. For a hard timeout, the CacheMaster sets a fixed timer; for a soft timeout, it periodically queries the traffic statistics to see if the rule has been inactive since the previous query.

Traffic statistics: 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. Upon receiving a `FlowStatRequest` query from the controller, CacheMaster can retrieve the latest counts from the switch that currently stores the rule, and can combine it with its own statistics. Note that CacheMaster also needs to query the traffic counters to update the weights needed by the cache-update algorithm[4] Depending on the frequency of queries from the controller, any polling CacheMaster does for the cache-update algorithm may also retrieve the statistics for the controller.

Barriers: On receiving a Barrier message, a switch must first complete all previous commands before executing any subsequent commands. CacheMaster also implements `Barrier` messages by making sure that the switch policy till a barrier is pushed to the caches before proceeding to the rules after the barrier. All the other messages (Handshakes, Config request/change messages) are relayed directly to the switch. Similarly, most of the other messages from the L_0 switch (PortStatus, Echo) are relayed directly to the controller.

Packet-in and Packet-out: Handling `packetIn` messages properly in the context of a multi-level cache requires some care because a `packetIn` from an L_i ($i > 0$) switch to the controller should have the L_0 inport in the message structure. In this case, the CacheMaster copies the inport that is encoded in the packet tag to the `PacketIn` message. If a `PacketIn` message comes from the L_0 switch, CacheMaster sends it directly to the controller. All `PacketOut` messages are sent to the L_0 switch directly.

5. Conclusion

In this paper, we have introduced CacheFlow, a new architecture and set of algorithms designed to support the abstraction of an infinite switch in software-defined networks. In order to make the abstraction efficient in the face of a large number of rules, CacheFlow's design is engineered to support the key properties of elasticity, transparency, fine granularity and malleability. As part of the design, we discuss the challenges posed by these properties and explain how to overcome them by exploiting the flexible network control interface that a Software-Defined Network provides.

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://www.sdncentral.com/products-technologies/sdn-openflow-tcam-need-to-know/2012/07/`, 2012.

[4] Measurement sources like NetFlow or sFlo, could be used instead, if available, to reduce the query overhead.

[3] B. Stephens, A. Cox, W. Felter, C. Dixon, and J. Carter, "PAST: Scalable Ethernet for data centers," in *ACM SIGCOMM CoNext*, Dec. 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 *IEEE ICNP*, (Washington, DC, USA), IEEE Computer Society, 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*, (New York, NY, USA), pp. 15–28, ACM, 2009.

[8] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: A GPU-accelerated software router," in *ACM SIGCOMM*, (New York, NY, USA), pp. 195–206, ACM, 2010.

[9] "Intel DPDK overview." See `http://www.intel.com/content/dam/www/public/us/en/documents/presentation/dpdk-packet-processing-ia-overview-presentation.pdf`, 2012.

[10] "The rise of soft switching." See `http://networkheresy.com/category/open-vswitch/`, 2011.

[11] N. Sarrar, S. Uhlig, A. Feldmann, R. Sherwood, and X. Huang, "Leveraging Zipf's law for traffic offloading," *SIGCOMM Comput. Commun. Rev.*, vol. 42, pp. 16–22, Jan. 2012.

[12] C. Kim, M. Caesar, A. Gerber, and J. Rexford, "Revisiting route caching: The world should be flat," in *Passive and Active Measurement*, (Berlin, Heidelberg), pp. 3–12, Springer-Verlag, 2009.

[13] D. Feldmeier, "Improving gateway performance with a routing-table cache," in *IEEE INFOCOM*, pp. 298–307, 1988.

[14] H. Liu, "Routing prefix caching in network processor design," in *International Conference on Computer Communications and Networks*, pp. 18–23, 2001.

[15] 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*, (New York, NY, USA), pp. 253–264, ACM, 2007.

[16] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with DIFANE," in *ACM SIGCOMM*, (New York, NY, USA), pp. 351–362, ACM, 2010.

[17] A. X. Liu, C. R. Meiners, and E. Torng, "TCAM Razor: A systematic approach towards minimizing packet classifiers in tcams," *IEEE/ACM Trans. Netw.*, vol. 18, pp. 490–500, Apr. 2010.

[18] S. Khuller, A. Moss, and J. S. Naor, "The budgeted maximum coverage problem," *Inf. Process. Lett.*, vol. 70, pp. 39–45, Apr. 1999.