Infinite CacheFlow : A Caching Solution for Switches in Software Defined Networks

by

Omid Alipourfard

A Thesis Presented in Partial Fulfillment
of the Requirement for the Degree
Masters of Science

Approved May 2014 by the
Graduate Supervisory Committee:

Violet R. Syrotiuk, Chair
Guoliang Xue
Andrea Werneck Richa

ARIZONA STATE UNIVERSITY

August 2014

ABSTRACT

New OpenFlow switches support a wide range of network applications, such as firewalls, load balancers, routers, and traffic monitoring. While ternary content addressable memory (TCAM) allows switches to process packets at high speed based on multiple header fields, today's commodity switches support just thousands to tens of thousands of forwarding rules. To allow for finer-grained policies on this hardware, efficient ways to support the abstraction of a switch are needed with arbitrarily large rule tables. To do so, a hardware-software hybrid switch is designed that relies on rule caching to provide large rule tables at low cost. Unlike traditional caching solutions, neither individual rules are cached (to respect rule dependencies) nor compressed (to preserve the per-rule traffic counts). Instead long dependency chains are "spliced" to cache smaller groups of rules while preserving the semantics of the network policy. The proposed hybrid switch design satisfies three criteria: (1)*responsiveness*, to allow rapid changes to the cache with minimal effect on traffic throughput; (2)*transparency*, to faithfully support native OpenFlow semantics; (3)*correctness*, to cache rules while preserving the semantics of the original policy. The evaluation of the hybrid switch on large rule tables suggest that it can effectively expose the benefits of both hardware and software switches to the controller and to applications running on top of it.

To my mother, brother and aunt . . .

ACKNOWLEDGEMENTS

I would like to thank ...

Violet R. Syrotiuk, for dealing with me through all this and for supporting me to the end. Daragh Byrne for supporting me through my first year at Arizona State University.

Jen Rexford, the person who broke most of the barriers that I thought were there for me, and showed me how all is possible. I want to thank her for graciously accepting me as a visiting student at Princeton University. Joshua Reich, one of the most awesome human beings that I know. He showed me how to care and how to think about a research problem.

Naga Katta, my good friend at Princeton University. He made my stay at Princeton a fun and a very rewarding experience. Srinivas Narayana, Cole Schlesinger and Laurent Vanbever, my friends at Princeton University, for patiently answering every one of my questions. Kelvin Zou, Nanxi Kang, Jennifer Gossels and the rest of my colleagues at Princeton for sharing their experiences with me.

My aunt, Sepideh Moghadam, for supporting me ever since I set foot in the US soil. She is the reason I am still moving forward. My uncle, Kaveh Faroughi, the guy that is always pushing the limits. David Birk, the kind man that showed me how to write and how to reason.

Finally, my mother and brother, the ones I love the most, which supported every one of my decisions. They are always there for me when I need them, and I would not have been here without their never ending support.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Chapter 1

INTRODUCTION

Computer networks are growing rapidly. New devices, services and applications are introduced to the network on a daily basis. With the addition of new entities, network operators must have finer-grained control over their traffic to better serve their customers. For example, operators use access control lists (ACLs) to provide security in the network. As new applications are developed, more sophisticated rules in the ACL are required to provide a safe network for customers.

Switches are network elements that enable a wide range of tasks such as packet forwarding and ACLs. However, these switches have small memory space available, which limits the granularity of the tasks that operators can define. We believe that as the networks grow and new paradigms, such as Software Defined Networking, are introduced the need for a switch with more memory becomes crucial.

The key contribution of this thesis is the design of a hybrid switch with a large amount of memory using commodity hardware and software that allows the higher demands of future networks to be met.

1.1 Switch Design Challenges

Today, a typical OpenFlow switch can process more than 100 million packets per second (Mpps) [9]. To achieve this speed, switches use "packet classification," a process that Gupta and McKeown [13] define as categorizing packets into "flows" that obey the same predefined rules. For every incoming packet, a switch has to find a rule among all rules that are installed in its rule table that match the packet headers, and execute the actions associated with that rule. A naive linear search through the rule

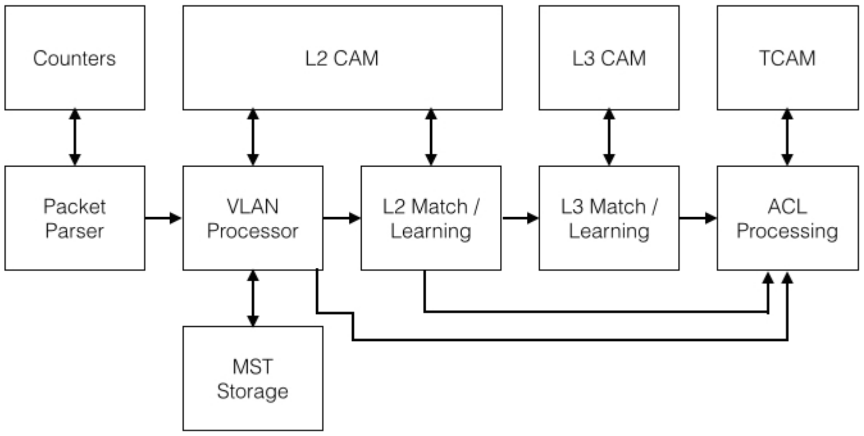


Figure 1.1: Input Processing Pipeline of A Switch [5].

table does not perform well. In fact, in the worst case on a switch with 5,000 rules (assuming a rule size of 320 bits) a bandwidth of 149 Tbps (100 Mpps $\times$ 5000 rules $\times$ 320 bits = 149 Tbps) between the processing unit and the memory unit is required to allow the switch to classify 100 Mpps. This bandwidth is far from being realizable on the current hardware. For comparison, the typical bandwidth between the external memory and the processing unit in a modern computer is only around 20 Gbps [27]. Fortunately, there are ways that allow us to perform better and reduce this bandwidth requirement, i.e., by using specialized hardware such as content addressable memory (CAM) and ternary content addressable memory (TCAM). In the next section, we explore how these hardware help to improve the performance of a switch.

1.1.1 A Closer Look at How Switches Work

Once a packet reaches a switch, the switch has to match the header of that packet against a rule table to identify what operations need to be executed on that packet. This is performed by passing the packet through a set of serialized data processing elements, which is also known as a processing pipeline that dictates how the incoming packets are handled, i.e., the outgoing port of the packet or the VLAN that the packets belong to is decided here. There are two main stages in this pipeline: input processing and output processing.

In the input processing stage, packets are not modified as any modification can affect the decisions made in later stages of this pipeline. In this stage, a set of actions in the form of metadata is attached to the packet. This is where it is possible to improve the lookup performance by using CAM or TCAM tables. Each of these tables has unique properties suitable for a particular set of actions. See sections 1.1.2 and 1.1.3.

Figure 1.1 shows CAM and TCAM tables in the input processing stage [5]. The first stage of pipelining involves checking the L2 CAM which matches against the packet's L2 header. Each rule in this table consists of a few bits. The L2 CAM table contains many such entries and therefore, is long and narrow. Next, the packet is matched against the rules stored in an L3 CAM table, which contains multicasting and Equal-Cost Multi-Path (ECMP) rules that match against the L3 header. These entries are typically larger than those in the L2 CAM, but at the same time are fewer in number. Finally, at the last stage of input processing the packet is matched against the TCAM table which is used for ACLs, and can only hold hundreds of entries.

After passing through the input processing stage, in the output processing stage the set of actions attached to the packet is executed. It is worth emphasizing that no new actions are attached to the packet in the output stage.

1.1.2 The CAM Table

CAMs are memory blocks that allow one to search for a piece of data in a single operation. This mechanism is very powerful, and to build up on the example

Header Field	**# of bits**
Ingress port	Implementation dependent
Ethernet Source	48
Ethernet Destination	48
Ethernet Type	16
VLAN ID	12
VLAN Priority	3
IP Source	32
IP Destination	32
IP Protocol	8
IP ToS bits	6
TCP Source Port	16
TCP Destination Port	16

Table 1.1: The Twelve Tuples in a TCAM Entry.

from section 1.1, the parallel search reduces the bandwidth requirement to 29.8 Gbps (100 Mpps $\times$ 320 bits = 29.8 Gbps); this is realizable on today's hardware. However, the width of the CAM table is decided at the time of design, and it only allows for matching on exact bits. This limits the usage of CAMs to MAC learning and multicasting. As an example, CAMs cannot be used for IP prefix matching since IP prefixes are of variable length. Fortunately, TCAMs allow for more general types of matching.

1.1.3 Switches Need TCAMs

TCAMs are memory blocks, which like CAMs, can be searched for a piece of data in one operation. The main difference between the two is that TCAMs allow for "don't care" bits in the data. This lends great flexibility and allows the TCAM to match on header fields that contain "don't care" bits; these fields are known as

Priority	**Rule**
10	`tcp-dest-port=http` → `forward`
9	`tcp-dest-port=ssh` → `forward`
1	`*` → drop

(a) Forwards HTTP and SSH traffic.

Priority	**Rule**
10	`tcp-dest-port=http` → `forward`
9	`tcp-dest-port=ssh` → `forward`
11	`*` → drop

(b) Drops all traffic.

Table 1.2: TCAM Rule Table.

wildcard fields. Typically, in a commodity, switch TCAMs are used for matching on the twelve tuples shown in Table 1.1. However, since TCAM allows for matching on wildcard bits, collisions can occur; therefore, a priority is assigned to each rule, so that in case of collision, only the rule with the highest priority is executed. The combination of wildcard fields and priority lists allow for complex dependency chains. As an example, Table 1.2a and Table 1.2b both contain the same set of rules, but with different priorities. In Table 1.2a, since the drop rule has the lowest priority, http and ssh packets are forwarded normally, but in Table 1.2b, because the drop rule has the highest priority, no packets ever reach the http or ssh rules, therefore, all packets are dropped regardless of the other two rules.

However, the flexibility of TCAMs comes at a price. Particularly, TCAMs have larger circuitry than static random access memories (SRAMs), occupying up to 40 times more die size. Also, TCAMs are 400 times more expensive, and exceptionally more power-hungry than SRAMs [4]. Some of these limitations can not be avoided even with the advances in technology. For example, the power consumption problem is inherent in the way the TCAMs work, i.e., a parallel search through all entries means that the TCAM's circuit is on at all times, and unfortunately this power consumption grows linearly with the size of TCAMs.

Furthermore, TCAMs operate at sub-gigahertz frequencies [14]. This means that

Rule	Source IP	Destination IP	Source Port	Destination Port	Protocol	Action
r_1	*	ip_1	[1,32766]	[1,32766]	UDP	drop
r_2	*	ip_2	[1,32766]	[1,32766]	UDP	drop
r_3	*	ip_3	[1,32766]	[1,32766]	UDP	drop
r_4	*	*	*	*	*	accept

Table 1.3: Access Control List Table.

as the switches become faster, the TCAMs needs to be replicated on the same die to keep up. Replication allows switches operating at gigahertz speed to distribute requests across sub-gigahertz TCAM banks without any performance penalties, but this replication also means that less die space is available per bank. The problem is exacerbated with the introduction of IPv6 because the size of each entry in the TCAM grows, which reduces the number of entries. Specifically, TCAMs can hold entries as wide as 640bits. Moving from IPv4 to IPv6 requires 96 additional bits per source and destination addresses. This is equivalent to $\frac{96*2}{640-256+64} = 42.85\%$ increase per entry size, which effectively reduces the total number of entries by 30 percent ($\frac{1}{1.4285} = 70\%$).

However, TCAMs are memory blocks that are required for the operation of a switch, particularly ACLs can only be implemented on TCAMs, and future networking paradigms, such as Software Defined Networking (SDN), make extensive use of this resource.

1.2 Today's Networking

Today, the primary use of TCAMs is in ACLs. ACLs are a means by which a switch identifies which packets should be forwarded and which ones should be dropped. Operators use these lists to restrict access to sensitive information within a network, or to mitigate distributed denial-of-service (DDoS) attacks. In these use cases, an ACL can grow at a very rapid rate. For example, consider a distributed denial of service attack on UDP ports 1 to 32766 that is targeting hosts with ip_1, ip_2, and ip_3. To mitigate such an attack, the operator might install ACL rules to drop all the UDP traffic to these ports and hosts. This ACL configuration is shown in Table 1.3. Due to range expansion [18], each of the first three rows in this table requires 784 TCAM entries; therefore, this table translates to $784 \times 3 + 1 = 2353$ TCAM entries, which is enough to fill the TCAM table of most commodity switches. Extra rules that do not fit in the ACL table, go through a software path which usually causes high CPU utilization on the switch and partially disrupts a switch's normal functionality. In fact, today, TCAM and ACL exhaustion are well known problems, and vendors such as Cisco have troubleshooting pages that suggest guidelines for avoiding TCAM exhaustion [7, 8].

In summary, if the network growth trend continues, operators will require additional TCAM space to store more sophisticated ACL rules to mitigate attacks and to better serve their customers; therefore, a solution that deals with the TCAM space issue is imperative.

1.3 TCAM and Software Defined Networking

The Gartner report names Software Defined Networking (SDN) as one of the emerging trends in information technology [28]. SDN separates the control and data plane to reduce the complexity of traditional networks. This separation provides strong abstractions and adds programmability to a distributed network. For instance, because of this abstraction, companies such as Google [15] and Microsoft [22] have adopted SDN in their data-centers to manage resources in a more efficient manner. OpenFlow [19] is a protocol that makes this separation possible. By using the Open-

Monitor
`srcip=5.6.7.8` → `count`

Route
`dstip=10.0.0.1` → `fwd(1)`
`dstip=10.0.0.2` → `fwd(2)`

Load-balance
`srcip=0*,dstip=1.2.3.4` → `dstip=10.0.0.1`
`srcip=1*,dstip=1.2.3.4` → `dstip=10.0.0.2`

Parallel Composition of Monitoring and Routing Policy
`srcip=5.6.7.8,dstip=10.0.0.1` → `count,fwd(1)`
`srcip=5.6.7.8,dstip=10.0.0.2` → `count,fwd(2)`
`srcip=5.6.7.8` → `count`
`dstip=10.0.0.1` → `fwd(1)`
`dstip=10.0.0.2` → `fwd(2)`

Sequential Composition of Load-balancing and Routing Policy
`srcip=0*,dstip=1.2.3.4` → `dstip=10.0.0.1,fwd(1)`
`srcip=1*,dstip=1.2.3.4` → `dstip=10.0.0.2,fwd(2)`

Table 1.4: Example of Composition of Several Policies [20].

Flow protocol, a central controller installs rules on switches around the network. However, since OpenFlow allows arbitrary wildcard fields in a rule, most of these rules can only be installed in the TCAM table. One of the many concerns regarding SDN is whether the current switches can store enough rules in the TCAM space to satisfy OpenFlow applications. To answer this question, we look at one of the promising features of SDN, namely "composition."

A potential benefit of SDN is providing a platform where independent software can coexist. This gives consumers great customizability for their network as they are free to select the software they need for their infrastructure and "compose" them together. In their work, Monsanto et al. [20] suggest operators that make it possible to run applications in sequence or in parallel.

To better describe these operators, consider three network applications for monitoring, routing, and load balancing. By using these three applications and the composition operators, it is possible to make more sophisticated applications. For example, if an operator wants to load balance the traffic across a set of servers, he can sequentially compose load balancing and routing applications. Or, for finding a congested link within the network, the operator might want to monitor the traffic without disturbing the routing policy. In this scenario he can compose routing and monitoring applications in parallel. Table 1.4 shows an example of the composition of these applications.

As seen in Table 1.4, the parallel composition of two policies creates many more rules, and in fact, composing two rules in parallel causes a multiplicative explosion in the number of rules. Therefore, while composition is a promising feature of SDN, it is far from being realizable considering the limited amount of TCAM space available on today's switches.

1.4 The Need for a Caching Solution

The need for more TCAM space is imperative in the current and future of networking. The solutions proposed in this space usually follow two general schemes, "caching" and "compressing" the rules in the rule table. Rule compression combines rules that perform the same actions and have related patterns [18]. 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, when compressing rules, we lose information on counters and timeouts of the original policy, which can provide vital information about the nature of the traffic in a network. Therefore, any solution should preserve the properties of each rule.

Internet traffic follows Zipf's law, i.e., a few rules match most of the traffic while the majority of rules handle the small portion of the traffic [23]. Based on this, we believe that caching is a reasonable alternative solution to the rule-space problem.

A caching scheme saves the most "popular" rules in the TCAM, and diverts the remaining traffic to a software switch (or software agent on the hardware switch) for processing. Our caching algorithm carefully splits the rules among software and hardware so that the semantics of the original policy are preserved. The combination of hardware and software gives the operator 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.

In order to make integration with existing networks easier, any good caching solution should have three properties:

Correctness: Caching rules should not change the overall policy in any manner. The rules should be cached very carefully so that the semantics of the original policy are preserved. Furthermore, caching should be done so that most of the network traffic is processed at line-rate.

Transparency: Entities that use the TCAM space should be oblivious to the existence of a caching layer; e.g., counters of rules should be updated in a consistent manner, and rules should timeout normally. Thus, any rule manipulation done by the caching abstraction should be transparent with respect to these expectations.

Responsiveness: A good caching solution should be dynamic, i.e., if a rule becomes popular during a certain time period, the caching solution should react in a timely manner, and move the rule in the cache hierarchy in order to minimize churn.

The rest of the thesis is organized as follows. Chapter 2 discusses recent caching and compression solutions for overcoming the TCAM space limitation problem. In Chapter 3, a caching system is proposed that satisfies correctness, responsiveness and transparency properties. Chapter 4 evaluates the system on few network policies. Finally, Chapter 5 discusses future work that can be studied in this space.

Chapter 2

RELEVANT WORK

The solutions proposed to manage the rule space problem generally fall into two main categories: caching and compression. Solutions that rely on compression, aim to make effective use of the available memory space by combining several rules together without affecting the semantics of the rule table. The problem with compression is that we lose information about rules, e.g., when two rules are merged, extracting the packet counter of each rule is not possible. This violates transparency, which is a desired property of any solution. On the other end, caching solutions usually break the rule table into several smaller rule tables, while preserving the semantics of the policy. These algorithms then save each of these rule tables based on the need of the network on fast memory, i.e., TCAM. Here, we look at a few solutions that have been proposed to make efficient use of available TCAM space: DIFANE [29], wire speed packet classification without TCAMs [10] and H-SOFT [11].

2.1 Scalable Flow-based Networking with DIFANE

In the early days of OpenFlow [19], solutions like Ethane [6] and NOX [12] sent the first packet of every flow to the controller. The controller then installed rules on switches in response to that packet. Unfortunately, sending the first packet of every flow introduced a lot of overhead on the controller, thus, this solution was not scalable. DIFANE proposed another solution in which packets not matching any rules on an ingress switch traversed a longer path through “authority” switches to reach their final destination. These authority switches then would first encapsulate the packet

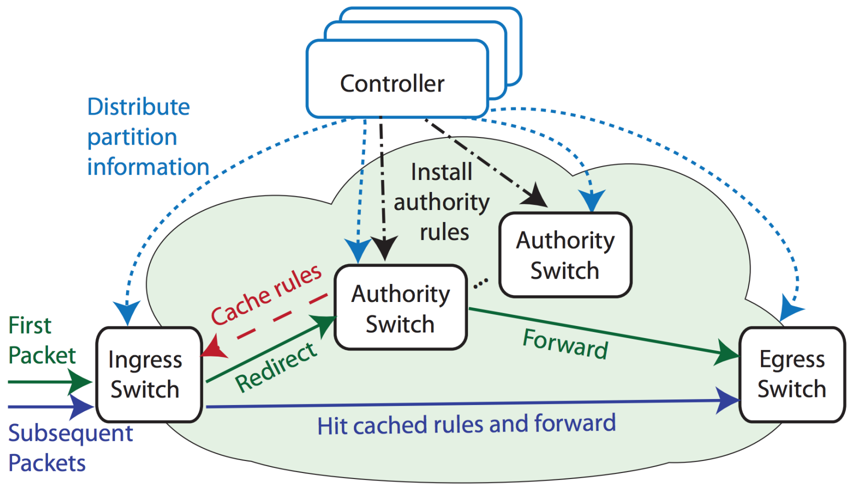


Figure 2.1: DIFANE Flow Management Architecture [29]. (Dashed Lines Are Control Messages. Straight Lines Are Data Traffic.)

and send it to their final destination, and second install a rule on the ingress switch. The newly installed rule will then forward all the incoming packets from the same flow to the corresponding egress port of the network. This way, DIFANE keeps all the packets on the data-plane and avoids sending unnecessary packets to a controller.

At the core of DIFANE lies an algorithm that carefully partitions the rule set across the authority switches, while preserving the semantics of the initial rule set. The number of partitions is equal to the number of authority switches available. The goal of the partitioning algorithm is to equally distribute the traffic among the authority switches and also, minimize the number of rules that will be split, i.e., a rule can extend across several partitions, in which case, the rule will be split into several rules, and each partition holds part of the initial rule. For example, for a rule table with rules $R_1, \ldots, R7$, and with pictorial projection [1] depicted in Figure 2.2,

[1]One can think of the header of a packet as a point in a discrete multidimensional space, where in this space each axis represents a field of the packet header. Since rules can contain wildcarded fields, each rule can encapsulate several of these points. The set of points in this space is known as the flow space of the rule. A rule table which contains several rule, has a projection in this multidimensional space which is referred to as pictorial projection.

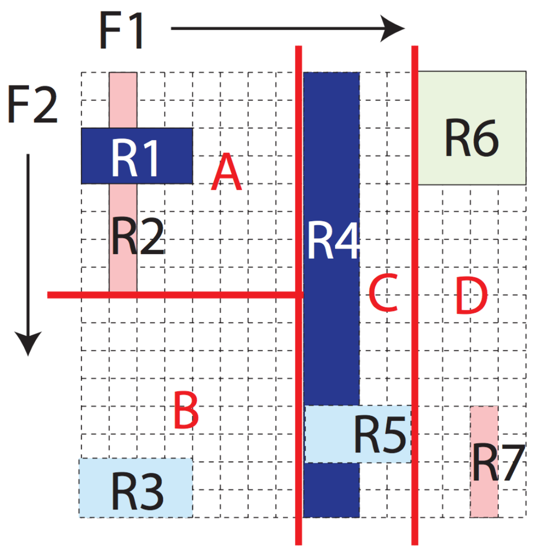


Figure 2.2: Low-level Rules and the Partition [29].

where each rule has two wildcard fields, F_1 and F_2, the DIFANE algorithm partitions the rule table into four different partitions, A, B, C, D and installs each partition on a separate authority switch.

For an incoming packet that lies in partition A, if the ingress switch has a rule for processing the packet, it will encapsulate the packet and send it to the corresponding egress switch. If the ingress switch does not have a rule for processing the packet, it would then forward the packet to one of the authority switches that manages partition A. The authority switch would then forward the packet and reactively install a rule on the ingress switch so that the rest of the packets of the flow are processed without hitting the authority switch. One can think of DIFANE, as a least recently used (LRU) caching scheme that reactively installs rules on new packets on the ingress switches while discarding the least recently used rules.

Since the rule sets in authority switches are a subset of the initial rule set, DIFANE ends up using more TCAM space than the initial rule set. Hence, DIFANE itself is a TCAM hungry solution that would benefit from a caching solution like ours.

Rule	Predicate and Action
I	$(F_1 \in [30, 70]) \wedge (F_2 \in [40, 60]) \rightarrow$ permit
II	$(F_1 \in [10, 80]) \wedge (F_2 \in [20, 45]) \rightarrow$ permit
III	$(F_1 \in [25, 75]) \wedge (F_2 \in [55, 85]) \rightarrow$ permit
IV	$(F_1 \in [0, 100]) \wedge (F_2 \in [0, 100]) \rightarrow$ deny

Table 2.1: A Rule Set of 4 Rules. Rules Ordered by Priority [10].

2.2 Wire Speed Packet Classification without TCAMs

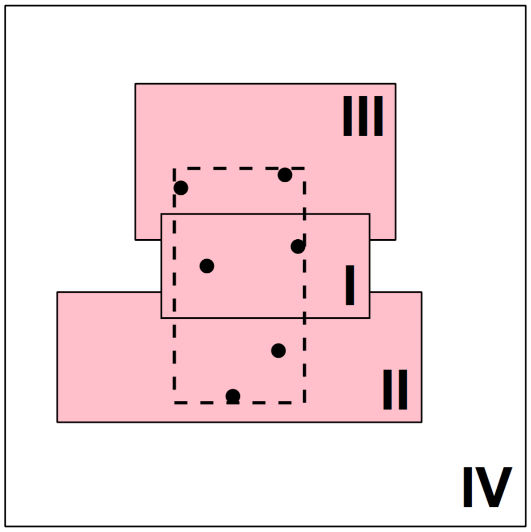


Figure 2.3: Caching an Independently Defined Rule Based on the Rule Set in Table 2.1.

Dong et al. [10] propose a hardware cache for solving the TCAM space problem. In their work, a software component creates a rule set based on the most popular rules, and saves it in the hardware cache. This rule set “evolves” with changes in traffic weights. For example, consider the rule set in Table 2.1, where each rule has two fields, F_1 and F_2 that can take a value between 1 to 100. The pictorial projection of this rule table is shown in Figure 2.3. Six flows, which are shown as dots in the

Figure 2.3, are passing through the router. As it can be seen, all of these flows are hitting one of the first three rules in the Table 2.1. The software component then creates a single rule, shown as the box with the dashed borders, that matches all of the six flows and saves it in the hardware cache. This new rule only requires one entry as opposed to three of the original policy. Note that this rule does not violate the policy, as any flows within the dashed box are processed by one of the first three rules which have the same set of actions as the cached rule. The rest of the traffic that is not matched by the rules in the cache is then processed by the software component. Evaluations suggest that by using the "evolving" cache, miss ratios that are 2 to 4 orders of magnitude lower than flow cache schemes are achieved [10]. Nevertheless, this solution suffers from the same problem as other solutions in the compression space, i.e., because reasoning about the rule counters is not possible, information on the nature of the traffic is lost, therefore, this solution is not transparent to the controller.

2.3 H-SOFT: A Heuristic Storage Space Optimization Algorithm For OpenFlow Tables

Finally, H-SOFT uses heuristics to decompose a rule table into several tables, i.e., a rule table that matches on n header fields will be decomposed into n tables where each table matches on a single header field. In the best case, decomposition can achieve a multiplicative decrease in the rule table size, that is a rule table with M rules and n fields can be decomposed into n rule tables, T_i with $|T_i|$ rules. Afterwards, by sequentially composing these rule tables in serial, i.e., connecting the output of the each table to the input of the next table, we can build the original policy, i.e.,

$$\prod_{i=1}^{n} |T_i| = M.$$

Unfortunately, the optimal rule table decomposition is NP-hard [24]. Also, the authors of H-SOFT do not take rule priorities into account, and because of this, their decomposition violates the semantics of the initial rule set.

2.4 Summary

While there have been novel solutions to provide more TCAM space to the controller that uses the switches, most of these solutions are not transparent to controller, and they affect the traffic distribution in unwanted ways. Solution like DIFANE introduce novel ways to "split" the rule table, but ends up using more TCAM space. Our solution uses a combination of rule splitting and software components to provide a transparent and responsive caching abstraction to controller and its applications.

To the best of our knowledge, our work in this thesis is the first study focusing on a caching solution that allows a large number of rules to be installed on a switch while preserving the semantics of the rule set and being transparent to the controller.

Chapter 3

PROPOSED SOLUTION

In this section, we introduce CacheFlow, a caching solution that aims to achieve the correctness, responsiveness and transparency properties which were identified in Chapter 1. The only requirement of CacheFlow is that the network should have separate control and data planes. This requirement leads us to design and test our system on top of SDN and OpenFlow.

OpenFlow uses a central controller that installs rules on the switches to manage the network. These rules are generated by applications running on the central controller, but since the limited rule space available on a switch is shared among all

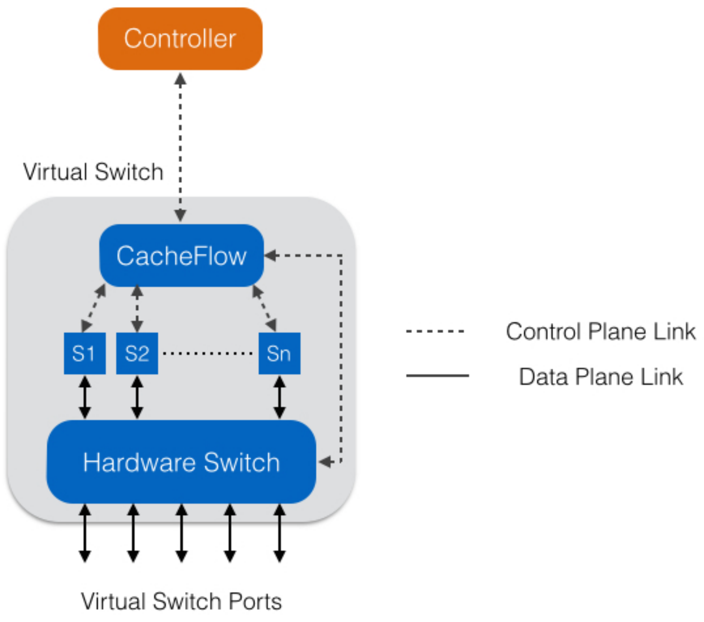


Figure 3.1: Architecture of CacheFlow.

such applications, it is very difficult for the central controller to efficiently manage this space. CacheFlow hides this rule space limitation from the controller and its applications, therefore enabling the controller to install, theoretically, infinitely many rules on the switches.

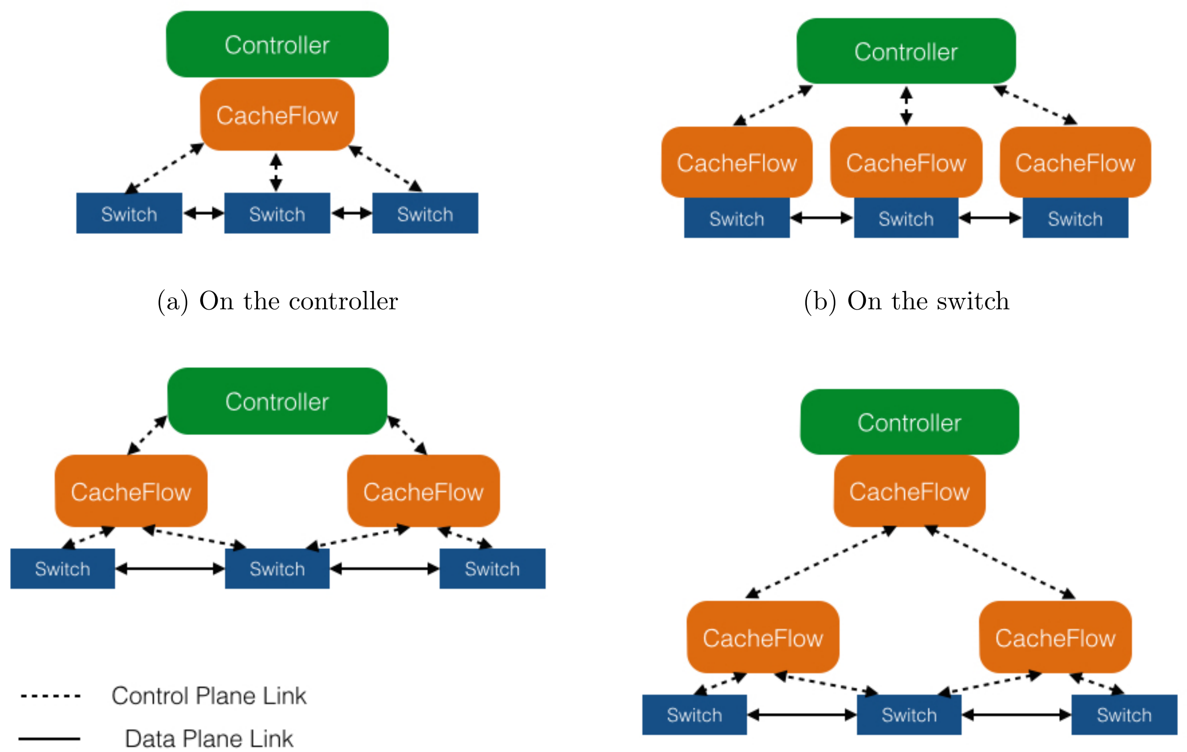
In order to provide a large rule space, CacheFlow makes a collection of one (fast) hardware switch and several slower switches (software, hardware or local agents) act like a single switch. The controller views this "virtual switch" as a normal switch with which it can communicate with using OpenFlow instructions. This virtual switch then distributes the rules among the underlying switches using the OpenFlow protocol. This architecture is shown in Figure 3.1. Since CacheFlow is transparent to the controller, it can be integrated into any system without modification.

Underneath, CacheFlow uses a dependency graph to manage the rule space and uses new algorithms in conjunction with this dependency graph to decompose the rule table of the virtual switch among multiple switches. Furthermore, this decomposition is done in such a manner that most of the network traffic passes through the (fast) hardware switch.

The other switches ($S_1, \ldots, S_n$ in Figure 3.1) together form a backup repository, where packets that experience a cache miss in the hardware switch are forwarded for processing. Thus, CacheFlow is purely a control-plane component (with control sessions shown as dashed lines), while OpenFlow switches forward packets in the data plane (as shown by the solid lines).

3.1 CacheFlow System

Figure 3.2 shows different possible configurations for CacheFlow deployment. We examine the four different deployment scenarios and compare their benefits and trade-offs.



(a) On the controller (b) On the switch

(c) In a separate box (d) A hybrid version

Figure 3.2: Design Choices for Placement of CacheFlow.

Deploying on the controller. The most accessible place to deploy CacheFlow is on the OpenFlow controller, as shown in Figure 3.2a. This gives CacheFlow a global view of the network, and allows CacheFlow to make network-wide decisions. For example, if a rule, R_1, is cached by a switch, it is arguably beneficial to cache R_1 on every other switch in the network to provide low latency to all the packets that hit R_1.

The problem with this approach is that deploying several instances of CacheFlow on the controller requires a significant amount of processing power. Therefore, the scalability of this solution is bounded by the processing power of the controller.

Deploying on the switch. Another possible scenario is to deploy CacheFlow directly on the hardware switches, as shown in Figure 3.2b. This approach has benefits compared to deploying on the controller, namely, it is much faster because CacheFlow has direct access to counters and timeouts, and requires minimal resources on the controller side. Also, because each switch has a CacheFlow instance running, this solution is not bounded by the processing power of the controller, consequently, scalability is simplified. Finally, since CacheFlow resides on the switch itself, it does not depend on the control plane protocol, i.e., it is not necessary to use OpenFlow in this scenario as CacheFlow has direct access to hardware.

The immediate problem with this configuration is that CacheFlow cannot optimize its decisions based on the available network-wide information. Also, this approach cannot scale beyond the processing power of the switch, which is naturally limited.

Deploying on a dedicated box. Another approach is to run CacheFlow on a separate box (Figure 3.2c). This configuration provides fault tolerance (since several instances of CacheFlow can manage the same switch), and scalability (since several switches can be using the same CacheFlow instance). The problem with this approach is that CacheFlow is tightly bound to the control plane protocol and lacks the global view that deploying CacheFlow on the controller provides.

Hybrid. A hybrid approach between the first and the third option allows CacheFlow to benefit from the global view of the network, and it also becomes scalable and fault tolerant, as shown in Figure 3.2d.

As we will see in Chapter 4, due to the simplicity of deploying CacheFlow on a controller platform, we chose the first configuration for our evaluations which is shown in Figure 3.2a. We implemented our system on top of Ryu, an OpenFlow controller, and use OpenVSwitch instances to evaluate CacheFlow.

3.2 CacheFlow Algorithm

In this section, we present CacheFlow's algorithm for placing rules in a TCAM with limited space. Since CacheFlow's algorithm runs in polynomial time, it can rapidly update the TCAM space, therefore, allowing CacheFlow to achieve responsiveness. CacheFlow then selects a set of "important" rules from the rules given by the controller, and caches them in the TCAM, while redirecting the cache misses to the software switches. Rules are split across TCAM and software switches so that the semantics of the overall policy are preserved. This allows CacheFlow to achieve correctness. CacheFlow also acts as a single OpenFlow switch, therefore it is transparent to the controller.

The input to the algorithm that CacheFlow uses to split the rule table, 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 if $i < j$. Each rule, R_i, also has a match, a set of actions, and a weight w_i that captures the volume of traffic matching the rule. The output is a prioritized list of k rules ($1 \leq k \leq n$) to store in the TCAM. CacheFlow aims to maximize the sum of the weights that correspond to "traffic hits" in the TCAM, while processing "all" packets according to the semantics of the original prioritized list. It is worth emphasizing that CacheFlow does *not* simply install rules on a cache miss. Instead, CacheFlow makes decisions based on traffic measurements over the recent past. 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.

3.2.1 Dependency Graph

The algorithm uses a dependency graph as the data structure for holding the rule table. By using this data structure, CacheFlow can efficiently split the rule table among switches. In what follows, we define a set of key concepts to allow for a more formal discussion.

Definition 1. A field, f, is a finite sequence of 0, 1 and 'x' ("don't care") bits.

Definition 2. A *predicate* is an n-tuple of (OpenFlow) fields. We use the notation f_i to access the *ith* field in the tuple.

This definition complies with how predicates are saved in a TCAM, i.e., if a predicate does not have a header field, it can be modeled as a predicate with a header field with a sequence composed of "don't care"s.

Definition 3. A *priority*, is an integer in the range of 0 to $2^{32} - 1$.

Definition 4. An action, a, specifies how a packet is processed in the pipeline, e.g., dropped or forwarded.

In this thesis, we ignore the semantics of an action and view it as a string. Two actions are equal if their strings are equal.

Definition 5. A rule, r, is a triple consisting of a priority, a predicate and a set of actions.

To access the fields in a rule r, we use the notation shown in Table 3.1.

Definition 6. A rule table, T, is an ordered list of rules, where the priority of rules in the list is in non-increasing order, that is:

$$\forall r_i, r_j \in T, \quad i > j \iff prio(r_i) \geq prio(r_j).$$

Function	Description
$pred(r)$	Returns *predicate* of rule r.
$prio(r)$	Returns *priority* of rule r.
$A(r)$	Returns the set of actions of rule r.
$h(p)$	Returns ph, the n-tuple of packet p's header fields.
$reg(f)$	Returns the regular expression associated with field f.

Table 3.1: Functions for Accessing Elements in Rules and Packets.

Definition 7. A packet, p, is a finite sequence of 0 and 1 bits.

Definition 8. A packet field, p_f, is a subsequence of bits within a packet.

Definition 9. A packet header, ph is an n-tuple of (OpenFlow) [1]. We use the notation ph_i to access the ith field in the tuple.

Definition 10. The regular expression of a field, $reg(f)$, is a regular expression in which each occurrence of 'x's in f is substituted with (0|1) expression.

For example, the field, $0x11x$ has a corresponding regular expression of the form 0(0|1)11(0|1).

Definition 11. A field matches a packet field if the regular expression of the field matches the packet field, i.e.,:

$$field\_match(f, p_f) \leftarrow p_f \in reg(f).$$

Definition 12. A rule, r, matches a packet, p, if:

$$m(p, r) \leftarrow (\forall i \implies (pred(r)_i \implies field\_match(pred(r)_i, h(p)_i))).$$

That is, the headers in predicate of the rule and the packet header are equal.

[1]Please note that packet field is different than a field. A field is a list of 0, 1, or 'x's whereas a packet field is a list of 0 and 1s

By using the above definitions we can talk about a rule that matches a packet in rule table, T.

Definition 13. Rule r_a in rule table, T, matches the packet, p, if:

$$(\forall r \in T, prio(r) > prio(r_a) \implies \neg m(p, r)) \wedge m(p, r_a).$$

Definition 14. A packet, p, "hits" a rule r in rule table T, if r matches p in T.

Definition 15. A rule r_a is shadowed by a rule r_b, if the priority of r_b is higher than the priority of r_a, and every packet that matches r_a also matches r_b, that is:

$$shadow(r_a, r_b) \leftarrow (\forall p, m(p, r_a) \implies m(p, r_b)) \wedge (prio(r_b) > prio(r_a)).$$

When a rule, r_a, is shadowed by another rule, r_b, no packets will ever hit r_a. Therefore, it can be safely removed from T without affecting the semantics of rule table, T.

By using the definitions above, we can now define the dependency between two rules, r_a and r_b in a rule table T.

Definition 16. Rule r_a depends on rule r_b if:

$$\begin{aligned} d(r_a, r_b) \leftarrow (\exists p, \forall r \in T, prio(r_a) < prio(r) < prio(r_b) \implies \\ \neg m(p, r) \wedge m(p, r_a) \wedge m(p, r_b)) \wedge \\ (prio(r_a) < prio(r_b)). \end{aligned}$$

That is, there exists a packet that is matched by r_b and r_a, but rules with priorities in between $prio(r_a)$ and $prio(r_b)$ do not match the packet.

Definition 17. The dependent set of rule r_a is a set of rules, $D(r_a)$, where for every rule, r, in $D(r_a)$ the property $d(r_a, r)$ holds, that is:

$$D(r_a) = \{r | (r \in T) \wedge d(r_a, r)\}$$

It is now possible to define a dependency graph given these definitions.

Definition 18. Given a rule table T, a dependency graph is a directed acyclic graph where the nodes correspond to the rules in the rule table T, and there exists an edge $e_{a,b}$ from rule r_a to r_b if the property $d(r_a, r_b)$ holds.

Theorem 1. *The dependency graph is acyclic.*

Proof. We prove by contradiction that the dependency graph is acyclic. If there is a cycle C in the graph with nodes $r_1, \ldots, r_n, r_1$, it means that there is an edge from $r_n \rightarrow r_1$, hence $prio(r_n) < prio(r_1)$, but it is also the case that $prio(r_1) < prio(r_2) < \cdots < prio(r_n)$. Due to total ordering of integers the $prio(r_1) < prio(r_2) < \cdots < prio(r_n) < prio(r_1)$ relation cannot hold. Therefore, the dependency graph is acyclic. □

In the next section, we first give a naive algorithm to build the dependency graph. After, we provide an incremental algorithm that builds upon the naive algorithm and is more efficient on machines with few processing cores. Also, in the next few sections, the arithmetic used for subtraction and intersection of predicates is based on the work of [16]. Below, you may find a short summary of the arithmetic of header space analysis for subtraction and intersection of predicates.

3.2.2 Arithmetic of Header Space Analysis

In the header space analysis [16], a predicate, also known as a wildcard expression, is a list of length L where each element can be 0, 1 or 'x' (wildcard), i.e., $\{0, 1, x\}^L$. Each wildcard expression can be viewed as a region in the hypercube with dimension L. A header space is the union of these wildcard expressions. The arithmetic defined here helps us to calculate the subtraction or intersection of these header space objects.

b_1 \ b'_1	0	1	x
0	0	z	0
1	z	1	1
x	0	1	x

Table 3.2: Intersection Rule for Bits.

Intersection: Two headers can have a non-empty intersection if they have the same value in every bit that is not a wildcard. Table 3.2 shows the intersection rule for the bits, b_1 and b'_1, that are wildcarded.

Union: Union of the header space objects creates a new header space object that has the wildcard expressions in both of the initial header space objects. For example, if h_1 and h_2 are two header space objects:

$$h_1 = \{xxxx0000, xx110001\}$$

$$h_2 = \{xxxx1111, xx110001\}$$

Then, the union of h_1 and h_2 is:

$$h_1 \cup h_2 = \{xxxx0000, xx110001, xxxx1111\}$$

Complementation: The complement of header h is the union of all headers that do not intersect with h, i.e.:

Subtraction: Subtraction of two header space objects is equal to the intersection of the first header and the complement of second header, that is:

$$h_1 - h_2 = h_1 \cap \bar{h_2}$$

```
$h' \leftarrow$
for bit $b_i$ in $h$ do
    if $b_i \neq x$ then $h' \leftarrow h' \cup x \ldots x\bar{b_i}x \ldots x$
    end if
end for
```

3.2.3 Static Algorithm for Building the Dependency Graph

The input to the "static" algorithm for building the dependency graph is a rule table, and the output is the dependency graph associated with the rule table. The most naive approach for creating the dependency graph is to iterate through all rule pairs in the rule table, and check whether Definition 16 holds. To verify Definition 16, we have to find a packet that can reach r_a after passing through the rules with priorities in between $prio(r_a)$ and $prio(r_b)$, or in the other words, the packet that "leaks" from r_b to r_a. To do that, we "subtract" the predicate of r_b and the predicate of rules in between r_a and r_b from the predicate of r_a. The subtraction gives a set of packet headers that will hit r_a but none of the rules in between r_a and r_b. The intuition behind this subtraction is that if CacheFlow installs r_a on a hardware switch then it also has to install r_b to preserve the semantics of the policy. Section 3.2.5 discusses why this installation preserves the semantics.

Since this algorithm runs on "snapshots" of a rule table, and the computation involved in this algorithm is independent of all the previous computations, we call it the "static" algorithm for building the dependency graph.

The static algorithm, which is shown in Algorithm 1, looks at every pair of rules (and the rules in between them) in a rule table with n rules. Therefore, it requires $O(n^2)$ checks to build the dependency graph. The strength of this algorithm comes from the fact that edges are computed separately, thus, it can be readily parallelized.

However, on a switch or a controller with a few processing cores, this algorithm does not perform well. In fact, short-lived rules cause rapid changes in a rule table, which in turn, cause a full recomputation of the dependency graph using the static algorithm. In the next section, we propose another algorithm that can incrementally modify the dependency graph to accommodate short-lived rules and rapid changes to the rule table.

Algorithm 1 Static Algorithm for Building the Dependency Graph

```
function STATIC-ALGORITHM-FOR-ONE-RULE($T$, $j$, $G$)
    packet_space = pred($T[j]$)
    for $j = j - 1$ to 1 do
        if packet-space $\cap$ $T[j]$ is not empty then
            $G[j][j]$ = packet-space
        else
            break
        end if
        packet-space -= pred($T[j]$)
    end for
end function
function STATIC-ALGORITHM($T$)
    $G[][]$ = empty
    % Iterate the rule table. $R_i$ is the $i_{th}$ rule in the table $T$.
    for $R_i$ in $T$ do
        % For each rule build the dependency graph
        STATIC-ALGORITHM-FOR-ONE-RULE($T$, $i$, $G$)
    end for
    return $G$
end function
```

3.2.4 *Incremental Algorithm for Building the Dependency Graph*

The input to the incremental algorithm is a rule table, T, a rule, R_j (that is going to be inserted into T or removed from it), and the dependency graph, G associated with the rule table. The output of the algorithm is a new dependency graph

Algorithm 2 Incremental Algorithm for Maintaining the Dependency Graph.

```
function FIND-AFFECTED-EDGES($T$, $j$, $G$)
    edges = hash(set())
    for $e_{i,j} \in G$ do
        between = $i < j \wedge \quad j > j$
        if between $\wedge (T[i] \cap \text{pred}(T[j])) \neq \emptyset$ then
            edges[$i$].push $j$
        end if
        return edges
    end for
end function
function UPDATE-AFFECTED-EDGES-WEIGHTS(edges, $T$, $j$, $G$)
    for key, edge-list $\in$ edges do
        % sort all edges by the priority of the parent
        edge-list = sorted(edge-list)
        % add the new edge between $R_1$ and $R_j$ as shown in Figure 3.3c
        $G[j]$[edge-list[0]] = $G$[key][edge-list[0]]
        for edge in edge-list do
            $G$[key][edge] = $G$[key][edge] - pred($T[j]$)
        end for
    end for
end function
function INCREMENTAL-ALGORITHM($T$, $j$, $G$)
    edges = FIND-AFFECTED-EDGES($T$, $j$, $G$)
    UPDATE-AFFECTED-EDGES-WEIGHTS(edges, $T$, $j$, $G$)
    STATIC-ALGORITHM-FOR-ONE-RULE($T$, $j$, $G$)
end function
```

corresponding to the new rule table.

First, we study how inserting a new rule affects the edges in the dependency graph. When inserting R_j into graph, G, it can affect the other edges in one of the several ways shown in Figure 3.3.

In the case that the priority of R_j is lower or higher than both R_1 and R_2, since the weight of edge $e_{1,2}$ depends only on the rules in between R_1 and R_2, addition of R_j will not affect $e_{1,2}$. This is shown in Figures 3.3a and 3.3b.

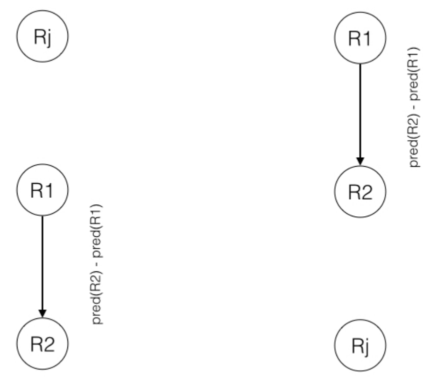
In the other case that the priority of R_j falls in between priority of R_1 and R_2, edge $e_{1,2}$ can change in one of the several ways. In the first case, if $pred(R_j) \cap pred(R_1) = \varnothing$, $e_{1,2}$ is not affected in anyway. Otherwise, if $e_{1,2} - pred(R_j) = \varnothing$, we can safely remove edge $e_{1,2}$, as R_1 is shadowed by R_j. This is shown in Figure 3.3c. Finally, as shown in Figure 3.3d, if $w(e_{1,2}) - pred(R_j) \neq \varnothing$, we only update the weight of $e_{1,2}$ to $w(e_{1,2}) - pred(R_j)$.

The core of the incremental algorithm is finding the edges that are affected by the addition of new rule R_j. Afterwards updating the weights of these edges is quite straightforward. Algorithm 2 shows a pseudo code of this approach.

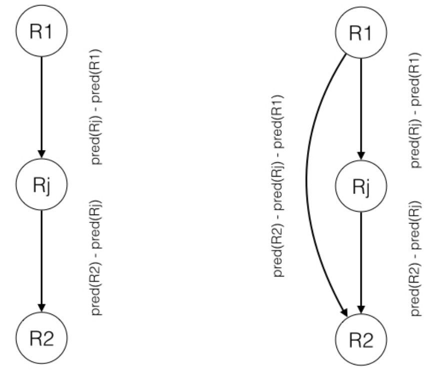
3.2.5 Dependent-Set, a Naive Algorithm for Splicing Dependency Chains

A simple straw-man approach allows us to build intuition about the problem. Using this intuition, we then proceed to find an optimal algorithm that avoids caching low-weight rules. From here on out, the straw-man algorithm is referred to as the "dependent-set" algorithm.

The input to the dependent-set algorithm is a rule table, T, a dependency graph, G, and a threshold. The threshold specifies the number of rules that can be safely installed in the TCAM. To maximize the traffic that is processed using the TCAM, we select rules that match the most traffic, and "cache" them in the TCAM. To preserve the semantics of the original policy, all the dependencies of the candidate rules should be cached in the TCAM as well. For example, consider the rule table shown in Figure 3.4. The dependency graph associated with this rule table is shown in Figure 3.5. On a switch capable of holding only one TCAM rule, because the weight of the rule R_6 is the highest, one might cache R_6 in the TCAM, and redirect the remaining traffic to the slower switches which hold the remaining rules. Unfortunately, only caching R_6 violates the semantics of the policy, as now part of the packets that



(a) $prio(R_j) > prio(R_2) > prio(R_1)$ (b) $prio(R_2) > prio(R_1) > prio(R_j)$



(c) $prio(R_2) > prio(R_j) > prio(R_1)$ (d) $prio(R_2) > prio(R_j) > prio(R_1)$

Figure 3.3: Affect of Inserting a New Rule into the Dependency Graph.

were hitting R_4, when storing everything in the same table, will now hit R_6. Therefore, it is not possible to only save R_6 in the TCAM. In this case, the solution is to either cache R_1 or R_2 as these rules do not depend on any other rules.

Now consider the case that the switch can hold four TCAM entries. Going back to Figure 3.4, because of the dependency between R_6 and R_4, when caching R_6, R_4 needs to be saved on the TCAM as well. However, there is also a dependency between R_4 and R_2. Therefore, to preserve the semantics when caching R_6, we should also

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 3.4: Example Rule Table.

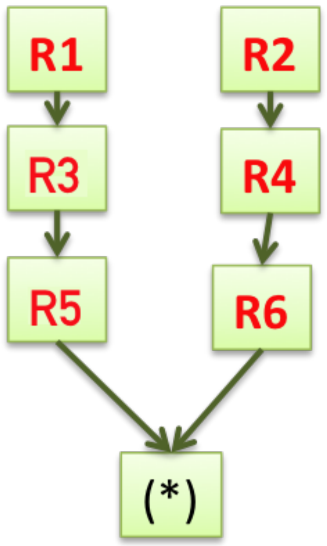
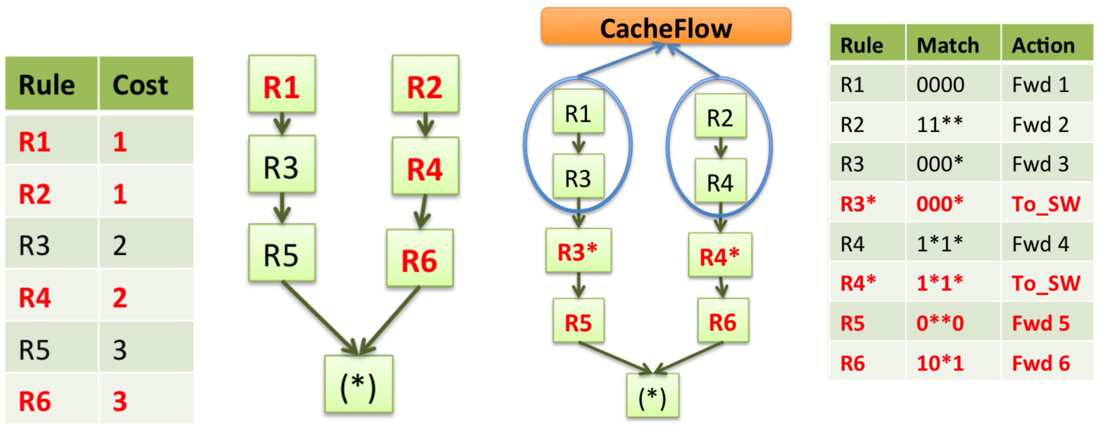


Figure 3.5: Dependency Graph Associated with the Rule Table in Figure 3.4.

cache two additional rules, R_4 and R_2. This implies that in order to preserve the semantics of our policy, all the rules in a dependency chain should be cached. In this case, the optimal rule set for caching is R_6, R_4, R_2, and R_1. In order to allow for a more precise discussion, we continue by first formalizing the problem statement.

In order to cache r_a, the set of dependents of r_a, $D(r_a)$, should also be cached. We refer to the union of r_a and $D(r_a)$ as $S(r_a)$. Therefore, to install r_a, at most $|S(r_a)|$



Rule	Cost
R1	1
R2	1
R3	2
R4	2
R5	3
R6	3

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

(a) Dependent-set Algorithm.

(b) Cover-set Algorithm.

Figure 3.6: Dependent-set vs. Cover-set Algorithms (L_0 Cache Rules in Red).

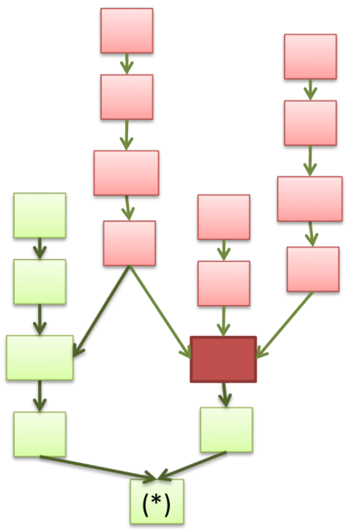
TCAM entries are needed. We refer to this number as the "cost" of rule r_a. Let $w(r_j)$ be the traffic that has been processed by rule r_j so far [2] . Also, let x_j be the indicator variable that denotes whether a rule r_j is installed in the TCAM or not.

The problem of choosing rules for installation in the TCAM, can be formulated as an integer linear programming (ILP) problem, i.e.:

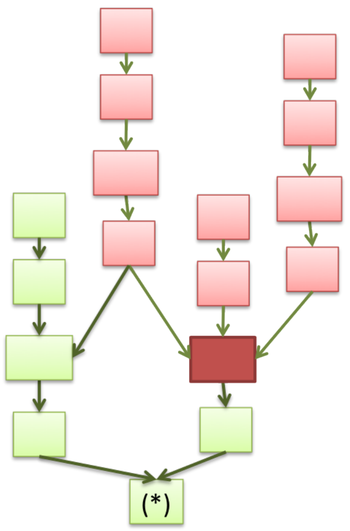
$$\begin{aligned}
\text{maximize} \quad & \sum_{r_j \in T} w(r_j) \cdot x_j \\
\text{subject to} \quad & \sum_{r_j \in T} x_j \leq \text{threshold} \\
& \sum_{r_j \in S(r_i)} x_j \geq |S(r_i)| \times x_i \\
& x_i \in 0, 1
\end{aligned}$$

The objective function is designed to maximize the traffic that is processed in the hardware switch. The first condition ensures that there is enough space in the TCAM

[2]This can be either the packet counter or the byte counter of rule r_j.



(a) Dependent-set Cost



(b) Cover-set Cost.

Figure 3.7: Dependent-set vs. Cover-set Costs (L_0 Cache Rules in Red).

for the chosen rules to be installed. The second condition specifies that if a rule, r_j, is chosen to be installed in the TCAM, all the dependent rules of r_j, $S(r_j)$, should also be installed in the TCAM.

Unfortunately, because an ILP problem is NP-hard, it is impractical to solve this problem on a controller. Therefore, we propose heuristics that gives a solution in $O(nk)$, where n is the total number of rules, and k is the number of entries in the TCAM. Our heuristic uses a greedy approach, where at each stage of the algorithm, a set, $S(r_a)$, with the highest aggregated weight to cost ratio, $w(S(r_a))/|S(r_a)|$, is chosen to be cached in the TCAM, this process is repeated until k rules are chosen for installation. This algorithm is shown in Algorithm 3. For example, for the rule table of Figure 3.4, with 4 available TCAM entries, the greedy algorithm would first choose R_6, R_4, and R_2, and then R_1, which brings the total cost to 4.

Algorithm 3 Dependent-Set Algorithm

```
function PREPROCESS-DEPENDENT-SET
    % Calculate the weights and costs according
    % to the dependent-set weight and cost.
end function
function DEPENDENT-SET(threshold, $T$, $G$)
    PREPROCESS-DEPENDENT-SET
    rules = $T$
    dep-rules = []
    while threshold $> 0$ do
        % Find the rule with the max weight
        candidate-rule = find-max($T$,key=weight/cost)
        candidate-dep-rules = candidate-rule.subgraph.rules
        if threshold $< 1 +$ candidate-dep-rules.length then
            % We do not have enough space to save this rule
            % ignore it in the next iterations.
            rules.remove(candidate-rule)
            continue
        end if
        dep-rules.push(candidate-rule)
        rules.remove(candidate-rule)
        for rule $\in$ candidate-dep-rules do
            dep-rules.push(rule)
            rules.remove(rule)
        end for
        threshold -= (1+candidate-dep-rules.length)
    end while
    return dep-rules
end function
```

3.2.6 Cover-Set, an Intuitive Approach for Splicing Dependency Chains

Respecting rule dependencies is not always feasible as it can lead to long dependency chains, i.e., a long chain of rules might need to be installed for caching a single rule. This can happen if a rule depends on many low-weight rules. For example, in a firewall with a single high-traffic, low-priority “accept” rule and many low-traffic, high-priority “deny” rules, caching the single accept rule requires installing many, if not all of the "deny" rules as well. Fortunately, it is possible to cache more efficiently by a variety of semantic-preserving methods. In particular, in this section, our algorithm "splices" the dependency chain by creating a small number of new rules that *shadow* many low-weight rules and sends the affected packets to the software switch.

For the example in Figure 3.6a, 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 3.6b.

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 for 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 3.6b; similarly, R_3^* is the cover-set for R_5. The rules defining these `forward_to_SW_switch` actions may also be merged, if necessary. The cardinality of the cover-set defines the new cost value for each chosen rule. 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 3.7a is 3 as shown in the rule cost table whereas the cost for the new cover-set for R_6 in Figure 3.7b is only 2 since we only need to cache R_4^* and R_6. Algorithm 4 shows an implementation of this cover-set approach.

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

3.3 Preserving OpenFlow Semantics

To preserve OpenFlow semantics, CacheFlow acts as a single OpenFlow switch by intercepting incoming OpenFlow messages and responding with appropriate actions. Table 3.3 shows the list of messages that the controller can send to an OpenFlow switch. For example, Section 3.2.5, 3.2.6 discussed algorithms that allow CacheFlow to correctly handle "Flow Modification" commands. For the rest of this chapter, we discuss how proper responses are generated based on the intercepted messages.

Flow Modifications. In the last section, we introduced two new algorithms for picking rules for installation in the TCAM. We also introduced a new data structure, the dependency graph, that could handle changes in the rule table, either statically or incrementally. When CacheFlow received a flow modification message, it first updates the dependency graph, and then picks a set of rules for caching in the TCAM by using either the dependent-set or the cover-set algorithms. Afterwards, CacheFlow sends a stream of flow modification messages to the hardware switch to update the cache, and also issues a flow modification message to every software switch to update the rule table of that switch.

Packet Outs. A packet-out message allows a controller to inject arbitrary packets into the network on a specific port of a switch. Since CacheFlow advertises hardware switch ports as the ports of a virtual switch, it is therefore enough for CacheFlow to change the port field of every incoming packet-out message to the corresponding hardware switch port. After, CacheFlow forwards the modified packet-out message to the hardware switch.

Port Modifications. Controllers manage the port behaviour using the port-mod messages. For example, by using port-mods, a controller can enable or disable the spanning tree algorithm on a specific port. Similar to packet-outs, CacheFlow preserves the semantics by forwarding these messages to the hardware switch after changing the virtual port field to the underlying hardware port number.

Barrier Requests. In the OpenFlow protocol, there are no guarantees on the execution order of received messages. Therefore, a controller uses a barrier-request message to ask the switch to finish processing all the previous messages before processing any new messages. After the switch has finished processing all the messages, it sends a barrier-reply back to the controller.

When CacheFlow receives a barrier-request message, it sends a barrier-request to all the underlying switches, and waits for the all the switches to reply back with a barrier-reply message. After receiving a barrier reply from every switch, CacheFlow then sends a barrier-reply back to the controller. This guarantees that all the pending messages have been executed before notifying the controller.

Flow Statistics Requests. A controller can request various statistics about a particular flow from the switch. These statistics include, duration, timeouts, packet-counts, and byte-counts.

Preserving duration is straightforward, as CacheFlow can save the initial time a flow was added to the rule table, and return the elapsed time, $T_{elapsed}$, to the controller.

To preserve rule timeouts, CacheFlow carefully calculates a "new" timeout when installing the rule on hardware or software switches. To calculate the new rule timeout, CacheFlow subtracts the initial timeout from the $T_{elapsed}$, and installs the rule with this new timeout.

#	OpenFlow Command
1	Flow Modification
2	Flow Statistics Request
3	Packet Out
4	Port Mod
5	Barrier Request

Table 3.3: List of OpenFlow 1.0 Switch Messages.

CacheFlow periodically polls the switches for rule packet-counts and byte-counts. These statistics allow CacheFlow to update the rule weights in the algorithms introduced in Section 3.2.5 and 3.2.6. Fortunately, CacheFlow can use the same flow statistics to reply back to flow-stat requests for byte-counts and packet-counts.

3.4 Summary

In this chapter, we introduced a dependency-graph data structure, which allows CacheFlow to *correctly* distribute the rule table across the switches. After, to improve the overhead of updating the dependency graph, we introduced an incremental-update algorithm to allow for rapid changes in the rule-table, therefore, we achieved *responsiveness*. Then, to transparently process OpenFlow messages, we introduced dependent-set and cover-set algorithms and also discussed how the rest of the OpenFlow messages are *transparently* processed.

Algorithm 4 Cover-Set Algorithm

```
function PREPROCESS-COVER-SET
    % Calculate the weights and costs according
    % to the cover-set weight and cost.
end function
function COVER-SET(threshold, $T$, $G$)
    PREPROCESS-COVER-SET
    rules = $T$
    cover-rules = []
    while threshold $>$ 0 do
        % Find the rule with the max weight
        candidate-rule = find-max($T$,key=weight/cost)
        star-rules = (candidate-node.children() - cover-rules)
        % if we have saved this rule as a star-rule,
        % it would not take any additional space in the TCAM
        % to overwrite the star-rule.
        space = 1
        if candidate-rule $\in$ cover-rules then
            space = 0
        end if
        if threshold $<$ space + star-rules.length then
            % We do not have enough space to save this rule
            % ignore it in the next iterations.
            rules.remove(candidate-rule)
            continue
        end if
        cover-rules.push(candidate-rule)
        rules.remove(candidate-rule)
        for rule $\in$ star-rules do
            cover-rules.push(rule)
        end for
        threshold -= (1+star-rules.length)
    end while
    return cover-rules
end function
```

Chapter 4

EVALUATION

We implemented a prototype [1] for CacheFlow in Python on top of the Ryu controller platform [26]. At the moment, the prototype transparently supports the semantics for the OpenFlow 1.0 features, except rule timeouts. We evaluated our prototype on top of an Open vSwitch as the software switch, and a Pica8 3290 as the hardware switch. This configuration is depicted in Figure 4.1.

As another experience, to evaluate the efficiency of the cover-set and dependent-set algorithms, we used two policies and assigned traffic weights based on the size of the flow-space, e.g., a rule that matches on source IP of 10.0.0.0/16 has 256 times more traffic than a rule that matches on source IP of 10.0.0.0/24. Then, we summed the traffic weights of the rules chosen by the algorithms. This sum is equal to the expected traffic-hit rate in the hardware switch. Table 4.1 shows a summary of the type of traffic and policies used in each experiment.

4.1 ClassBench Experiment

The first policy is a synthetic Access Control List (ACL) generated using ClassBench [25]. The policy has 10K rules that match on the source IP address, with

Experiment	Policy used	Traffic Trace Available	# of packets	# of rules
Synthetic	Stanford Backbone	✗	N/A	$1,836$
	ClassBench	✗	N/A	$10,000$
Hardware-OVS	REANNZ	✓	$3,899,296$	456

Table 4.1: Experiments Summary.

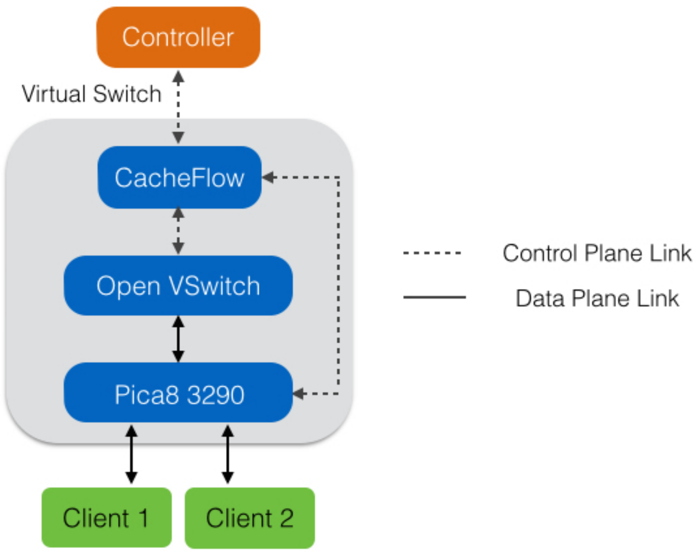


Figure 4.1: CacheFlow Configuration (Pica8 3290 - OVS).

long dependency chains of maximum depth 10. In the absence of a traffic trace, we created a dependency graph, and assigned traffic to each rule proportional to the portion of flow space it matches. Afterwards, we ran the cover-set and dependent-set algorithms, and measured the traffic that was being matched on the TCAM; that is, we summed up the traffic of the rules that were selected for caching.

Figure 4.2 shows the cache-hit percentage across a range of TCAM sizes, expressed relative to the size of the policy, e.g., for TCAM Cache Size of 1%, we assumed that the TCAM has space for $10000 * 1\% = 100$ entries. The cover-set does better than the dependent-set algorithm, mainly because of the long dependency chains in the ClassBench policy, which causes the dependent-set algorithm to install a long chain of rules, whereas the cover-set can install a few rules for each of the long rule chains. While cover-set has a hit rate of around 87% for 1% cache size (of total rule-table), both algorithms have around a 90% hit rate with just 5% of the rules in the TCAM. It is worth noting that since this experiment is deterministic, i.e., subsequent runs of

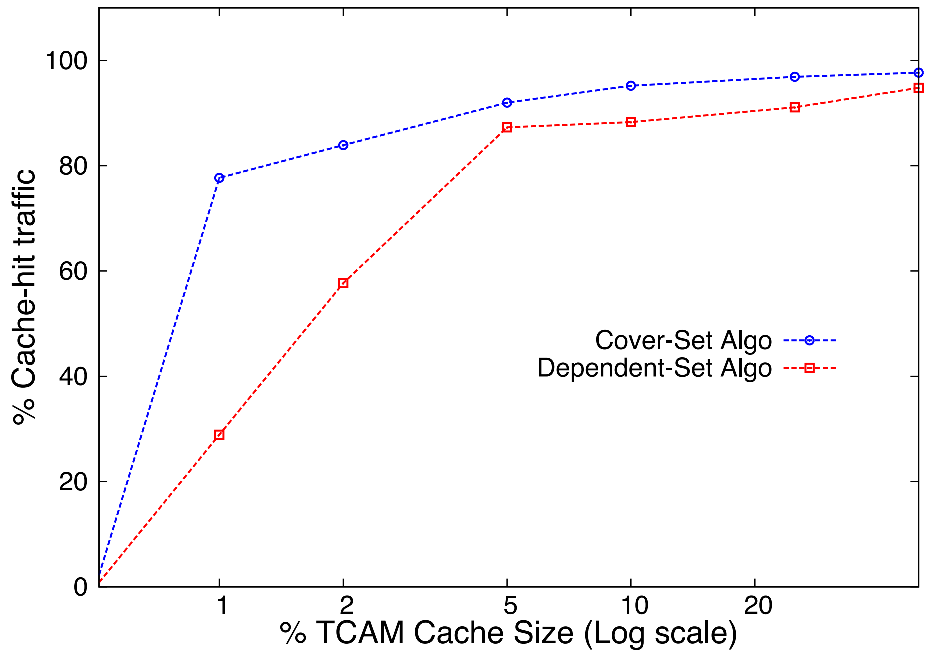


Figure 4.2: ClassBench Results.

the experiment give the same result, therefore, the algorithms were only run once.

4.2 Stanford

Figure 4.3 shows the results for a real-world Cisco router configuration on a Stanford backbone router [3]. This policy matches on pairs of source and destination IP addresses. Therefore, to transform the policy into an equivalent OpenFlow policy we used the `nw_src` and `nw_dst` fields specified in the OpenFlow specification, to match on source and destination IP addresses. This translation resulted in $1,836$ OpenFlow 1.0 rules that match on the source and destination IP address, with dependency chains with at most 6 levels of depth. Similar to the ClassBench experiment, We analyzed the cache-hit ratio by assigning traffic volume to each rule proportional to the size of

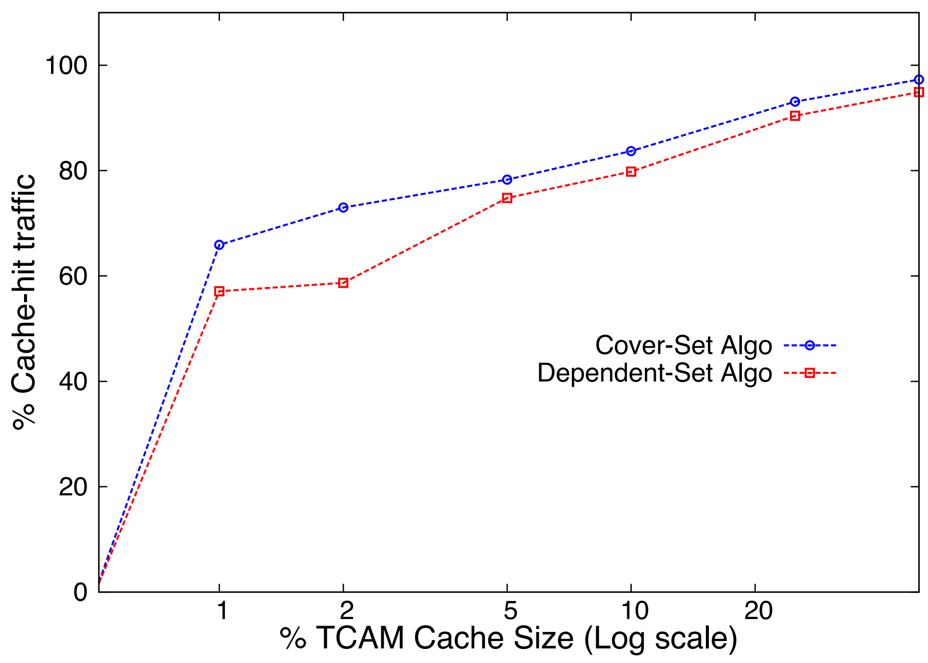


Figure 4.3: Stanford Results.

its flow space. The poor performance of dependent-set algorithm is explained by the fact that this policy, similar to ClassBench, has long dependency chains, which forces the dependent-set algorithm to cache all the rules in of a chain in the TCAM.

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. In this experiment, our algorithms did worse than the ClassBench case; this performance degradation is due to the fact that the flow space of each rule in the Stanford policy has a similar size, i.e., the traffic is distributed equally across the rules. Because of this similarity in size, every rule is as important as other rules with similar flow-space sizes to be cache in the TCAM. Similar to the ClassBench experiment, because of the deterministic nature of the experiment, we only ran the experiment once.

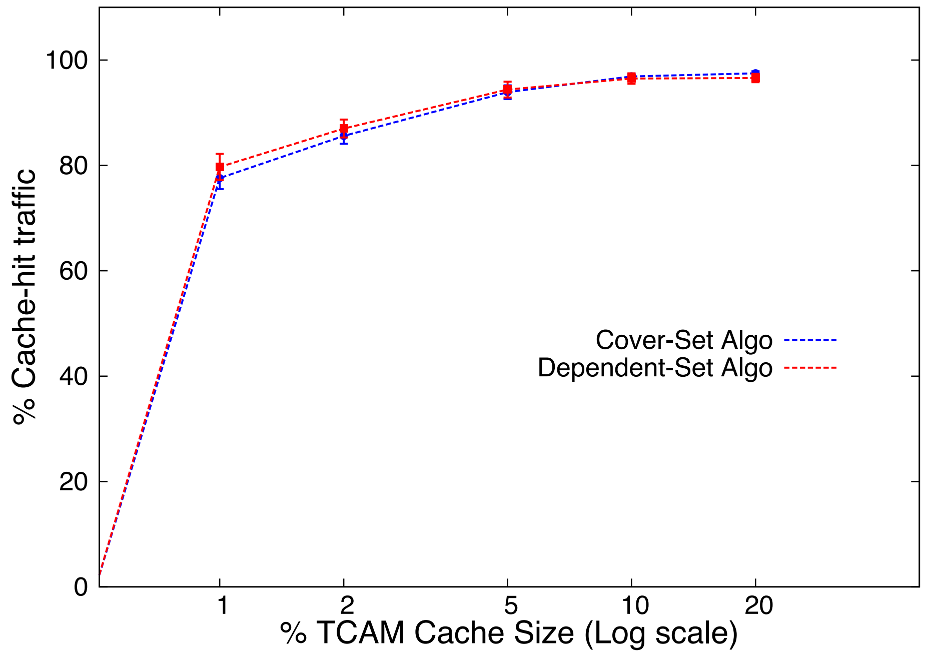


Figure 4.4: REANNZ Results.

Figure 4.4 shows results for the forwarding policy on an SDN-enabled Internet eXchange Point (IXP) in New Zealand that supports the REANNZ research and education network [2]. This real-world OpenFlow policy has 456 Openflow 1.0 rules matching on multiple packet headers like `inport`, `dst_ip`, `eth_type`, `src_mac`. The dependency graph of this policy has mostly dependencies of depth 1. As a result, we expected to see similar performance across all three rule-caching algorithms.

We replayed a traffic trace that we logged for two days to understand how a periodic cache update affects the cache-miss rate. We updated the cache using dependent-set and cover-set algorithm every two minutes and measured the cache-hit rate over the entire two-day period. Because of the shallow dependencies, all three algorithms have the same performance. All the algorithms see a cache hit rate of more than 80% with a hardware cache of just 2% of the rule table size. With just 10% of the rules, the average cache hit rate increases to as much as 97%. This experiment was performed 5 times to reduce the effect of the start time of replaying the traffic on the software and hardware switches.

To check the correctness of our prototype, we looked at rule counters in the absence and presence of CacheFlow and confirmed that CacheFlow is not affecting the counters. During the experiment, we did not modify any of the applications running on the controller, e.g., RouteFlow [21]. This suggests that CacheFlow is a transparent layer that does not affect the controller or the applications running on top of it.

4.4 Summary

Our evaluations on synthetic policies, namely Stanford and ClassBench, suggest that we can achieve very high cache hit rates by putting few rules in the TCAM of the hardware switch. We confirmed this hypothesis by running the experiments on a collection of a hardware switch and a software switch, by using a real network policy and traffic trace.

Chapter 5

CONCOLUSION AND FUTURE WORK

In this thesis, we discussed the need for a caching solution for switches in a network, then continued by talking about the properties of a good caching solution. Based on these properties, we designed CacheFlow, a caching system to enable fine-grained policies in Software-Defined Networks by optimizing the use of the limited rule-table space in hardware switches. CacheFlow achieves this by dividing the rule table so that most of the traffic is processed on the hardware switch, while the rest of the traffic goes through one or more slower switches. Several deployment configurations for CacheFlow were proposed, and we discussed how cache misses are handled in each of them: (i) one or more software switches to keep packets in the "fast path," at the expense of introducing new components in the network, (ii) in a software agent on the hardware switch to minimize the use of link bandwidth, at the expense of imposing extra CPU and I/O load on the switch, or (iii) 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.

Two algorithms (dependent-set and cover-set) and a data structure (dependency graph) are suggested that make CacheFlow possible. As part of future work, we believe that the dependency graph introduced in this work can be effectively used for other purposes as well, e.g., Net Plumber [17], removing shadowed rules, caching, minimizing priority lists are a few other applications that may benefit from this dependency graph.

Note that CacheFlow is by no means a final solution to the TCAM limitation problem and there are still a wide range of questions that can be asked, e.g., multi-table caching, global caching behaviour, and caching in conjunction with compression.

Multi-table Caching. As of now, CacheFlow can only manages a single rule table, but in OpenFlow 1.1+, there are instructions for creating multi-table packet processing pipelines. This raises the question of how a caching solution would look for a multi-table pipeline, e.g., is it more efficient to cache each table individually? How is a cache miss handled in this pipeline? Is it beneficial to move cache misses to the beginning of a multi-table pipeline?

Global Caching Behaviour. CacheFlow only manages the space of one switch. One might argue that a global caching solution, which takes the network-wide state into account when deciding whether to cache a rule or not, can provide lower latency to the end users and improve overall user experience; this configuration is shown in Figure 3.2a.

Caching and Compression. One can also consider "caching" and "compression" together, i.e., is it possible to compress and cache rules and satisfy responsiveness, transparency and correctness?

In light of the lessons learned in this thesis, we believe a rule-caching system is a viable solution to the memory limitation problem in switches around the network. Our caching solution provides a strong abstraction from the underlying hardware, and guarantees a uniform behavior across the switches. This is done by exposing a virtual switch interface which allows controllers to install a large number of rules regardless of the switch type. Further analysis is needed to quantify the extent to which one can exploit CacheFlow and its effect on network performance.

REFERENCES

[1] Cacheflow prototype. https://bitbucket.org/nkatta/cacheflow.

[2] REANZZ. http://reannz.co.nz/.

[3] Stanford backbone router forwarding configuration. http://tinyurl.com/mcnq57t.

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

[5] Marshall Brinn. GEC16: OpenFlow Switches in GENI. See https://www.youtube.com/watch?v=RRiOcjAvIsg, 2013.

[6] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Natasha Gude, Nick McKeown, and Scott Shenker. Rethinking enterprise network control. *IEEE/ACM Transactions on Networking*, 17(4), August 2009.

[7] Cisco. ACL and QoS TCAM Exhaustion Avoidance on Catalyst 4500 Switches. http://www.cisco.com/c/en/us/support/docs/switches/catalyst-4000-series-switches/66978-tcam-cat-4500.html, 2005.

[8] Cisco. Understanding ACL on Catalyst 6500 Series Switches. http://www.cisco.com/en/US/products/hw/switches/ps708/products_white_paper09186a00800c9470.shtml, 2010.

[9] Cisco. Cisco Catalyst 4500 Series Switches. http://www.cisco.com/c/en/us/products/switches/catalyst-4500-series-switches/models-comparison.html, 2014.

[10] Qunfeng Dong, Suman Banerjee, Jia Wang, and Dheeraj Agrawal. Wire speed packet classification without TCAMs: A few more registers (and a bit of logic) are enough. In *ACM SIGMETRICS Performance Evaluation Review*, volume 35, pages 253–264. ACM, 2007.

[11] Jingguo Ge, Zhi Chen, Yulei Wu, et al. H-soft: a heuristic storage space optimisation algorithm for flow table of OpenFlow. *Concurrency and Computation: Practice and Experience*, 2014.

[12] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: Towards an operating system for networks. *SIGCOMM CCR*, 38(3), 2008.

[13] Pankaj Gupta and Nick McKeown. Packet classification on multiple fields. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '99, pages 147–160, New York, NY, USA, 1999. ACM. ISBN 1-58113-135-6. doi: 10.1145/316188.316217. URL http://doi.acm.org/10.1145/316188.316217.

[14] Renesas Electronics America Inc. 20Mbit QUAD-Search Content Addressable Memory. See `http://www.renesas.com/media/products/memory/TCAM/r10pf0001eu0100_tcam.pdf`, 2010.

[15] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a Globally-deployed Software Defined WAN. *SIGCOMM Comput. Commun. Rev.*, 43(4):3–14, August 2013. ISSN 0146-4833. doi: 10.1145/2534169.2486019. URL `http://doi.acm.org/10.1145/2534169.2486019`.

[16] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=2228298.2228311`.

[17] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, pages 99–112, Berkeley, CA, USA, 2013. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=2482626.2482638`.

[18] Alex X. Liu, Chad R. Meiners, and Eric Torng. TCAM Razor: A systematic approach towards minimizing packet classifiers in TCAMs. *IEEE/ACM Trans. Netw*, April 2010.

[19] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM CCR*, 38(2):69–74, 2008. doi: http://doi.acm.org/10.1145/1355734.1355746.

[20] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software-defined networks. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, pages 1–14, Berkeley, CA, USA, 2013. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=2482626.2482629`.

[21] Nascimento, Marcelo R. and Rothenberg, Christian E. and Salvador, Marcos R. and Corrêa, Carlos N. A. and de Lucena, Sidney C. and Magalhães, Maurício F. Virtual Routers As a Service: The RouteFlow Approach Leveraging Software-defined Networks. In *Proceedings of the 6th International Conference on Future Internet Technologies*, CFI '11, pages 34–37, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0821-2. doi: 10.1145/2002396.2002405. URL `http://doi.acm.org/10.1145/2002396.2002405`.

[22] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu,

Changhoon Kim, and Naveen Karri. Ananta: Cloud scale load balancing. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 207–218, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2056-6. doi: 10.1145/2486001.2486026. URL `http://doi.acm.org/10.1145/2486001.2486026`.

[23] Nadi Sarrar, Steve Uhlig, Anja Feldmann, Rob Sherwood, and Xin Huang. Leveraging Zipf's law for traffic offloading. *SIGCOMM Comput. Commun. Rev. 2012*.

[24] Subhash Suri, Tuomas Sandholm, and Priyank Warkhede. Compressing two-dimensional routing tables. *Algorithmica*, 35(4):287–300, 2003.

[25] David E. Taylor and Jonathan S. Turner. Classbench: A packet classification benchmark. *IEEE/ACM Trans. Netw.*, 15(3):499–511, June 2007. ISSN 1063-6692. doi: 10.1109/TNET.2007.893156. URL `http://dx.doi.org/10.1109/TNET.2007.893156`.

[26] Nippon Telegraph and Telephone Corporation. Build SDN Agilely. `http://osrg.github.io/ryu/`, 2013.

[27] Wikipedia. DDR3 SDRAM. `"http://en.wikipedia.org/wiki/DDR3_SDRAM"`, 2014.

[28] Network World. Gartner: The Top 10 IT altering predictions for 2014. `http://www.networkworld.com/news/2013/100813-gartner-predictions-274636.html`.

[29] Minlan Yu, Jennifer Rexford, Michael J. Freedman, and Jia Wang. Scalable flow-based networking with DIFANE. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 351–362, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0201-2. doi: 10.1145/1851182.1851224. URL `http://doi.acm.org/10.1145/1851182.1851224`.