VERIFLOW SYSTEM ANALYSIS AND OPTIMIZATION

BY

ZHONGBO CHEN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Advisor:

Assistant Professor Matthew Caesar

Abstract

Computer Networks are complex and prone to bugs. Most existing tools designed to fix these problems run offline and can only fix the problems after they occur. Those tools often run at the timescale of seconds to hours. With the advent of Veriflow [7], network administrators are able to do real-time checking of network-wide invariants.

VeriFlow is an efficient tool designated to detect SDN network invariants at run time. It serves as an intermediate layer between software defined networking controller and the actual network devices. When each forwarding rule is to be inserted, modified or deleted into the network, the Veriflow will check for network-wide invariants violation dynamically before an actual action would take place.

The Veriflow system can perform checking within hundreds of microseconds per rule. The invariant checking process contains three major phases: 1. Packet lookup phase—giving a new rule change, find all rules that will be affected given the insertion, deletion or modification of this new rule change. 2. Forwarding Graph Construction phase—Construct forwarding graph of those rules affected. 3. Graph Traversal phase—traversing those forwarding graphs to detect problems.

In this work, we aim to boost the performance of the Veriflow system because the original system might fail to meet the performance requirements in some aspects. To achieve that, we first performed an experimental study of the system performance to figure out the where the bottleneck is in each of the following three phases. In phase 1, we surveyed a couple alternative lookup data structures and decided to adopt a multi-level red-black tree based solution. We also proposed a new way to find equivalent class based on our new lookup data structure and a new IP prefix format. In phase 2 and 3, we embedded the graph construction into rule insertion and store the graph in the lookup data structure directly to reduce the overhead. In addition, we did quite a few software engineering related optimizations to further reduce the running time and revamped the code base. We feed the same dataset to both implementations and our implementation will always have a consistently high speedup. This speedup grows linearly with the increases in network complexity. The new code base that comes along with our implementation is also better structured and more readable.

To Father and Mother.

Acknowledgments

I'd like to thank my academic advisor, Matthew Caesar, for his mentorship and advices that make this research possible. I learnt a lot from him for his quick insight into problems, attention to details, and a strong pursuit for better solutions. I am delighted to have him as my advisor.

I'd like to thank Danny Xu for all of his efforts put into this project as a partner. He is a person who is willing to work really hard to get things done. During our cooperation on this project, he could always give me useful advice and we can always discuss interesting ideas that lead to better, simpler solutions.

Table of Contents

Chapter 1

Introduction

The classical network is built on top of a single common communication protocol, IP. The Internet is composed of many autonomous systems managed by individual Internet Service Providers. Virtually, every ISP deploys the network using network devices manufactured by proprietary vendors, runs their own proprietary protocol on top of those devices, and leaves little room for flexibility. Even inside the same ISP, it might run different protocols on different ASes due to technical issues or external influences, such as local competitions or collaborations.

The clear lack of industry standard makes the Internet largely proprietary nowadays: network devices are often designed to run specific proprietary protocol, leaving out general functionality for sake of cost and performance. The economic efficient solution, however, encapsulates the device internal, making it hard to customize network policies since it tightly couples the forwarding plane with data plane.

Moreover, packet forwarding in modern network systems has always been the most sophisticated and error-prone process. Hundreds or thousands of network devices, such as routers, switches, and firewalls are all connected together making the network topology huge and complex. In addition, those network devices composing the network topology are usually made from different vendor. The complexity of the topology itself and the heterogeneity of device manufactures lead to a substantial amount of efforts being paid to ensuring the network security and correctness. However, faults still occur in various forms in the real world, such as loops, black holes, to name just a few.

Software Defined Networks, SDN, was introduced as a solution. The idea is to separate forwarding plane with data plane by abstracting the flow control, letting it to be managed by a logically centralized controller. This separation allows large-scale deployment of new flow management/traffic engineering protocols, allowing more rapid adaption of new ideas in networking research. In addition, it allows network administrators to define behavior of the network programmatically, making dynamic allocation of tasks to devices possible.

OpenFlow [8] is the de facto protocol that is running on top of SDN. There have already been a couple of large-scale applications based on SDN built, such as B4 [5]. However, while the introduction of SDN eases network administrators by providing them with an interface to program network, potential errors in packet forwarding might still occur if there are several conflicting rules inserted when multiple network administrators are modifying them at the same time. Or if it is just a sub-optimal policy. There is always a need for an efficient online debugging tool for packet forwarding.

In the past, most users have to check at network snapshots to find bugs but this can be very inefficient and can only be done offline. Methods applied to small-scale system such as symbolic execution, which explores all possible path of a program execution, also does not apply to problem of this large scale. Veriflow [7] is born out of this requirement. It is a real-time network-checking tool designed for networks deployed in SDN environment.

Having carefully examined the source code of VeriFlow, we noticed that we can make some significant optimizations to it. This thesis will survey the original Veriflow, presents and finally evaluates the several optimizations made.

1.1 Contributions

In this work, we made the following major contributions:

1. Surveyed the overall Veriflow [7] code base. Benchmarked each component of the Veriflow using Intel V-tune [13] to find out the bottlenecks of each.

2. Surveyed and changed the lookup data structure from a Single-Bit Multi-level Trie based structure to a Multi-level RB-Tree based structure. Defined a new IP format representation within the data structure and refined the concept of Equivalence Class. Created a new algorithm to do packet search and equivalence class cut, which resulted in much fewer forwarding graphs. The result of this contribution is a significant improvement in the packet search time by 7x.

3. Simplified the processes of constructing and traversing forwarding graphs. Reduced unnecessary overhead between these processes and used a new algorithm to traverse forwarding graphs. The result of this contribution is a combination of graph construction and traversal phase and another cut in overall checking time by 4x.

4. Cleaned up the code base and refactored a large portion of the critical functions. Adopted Intel V-tune [13] to investigate into the bottleneck of each function and implemented many software engineering related optimizations, such as reducing unnecessary classes and structures, making code more readable, etc. The result is a more than 4x reduction in lines of code and some further improvement in performance.

Chapter 2

Study of Veriflow System

2.1 Background: Veriflow General Introduction

The emergence of SDN provides us with excellent opportunities to innovate and create novel features in network design and network testing, given the existence of a centralized controller that has a global understanding of the underlying network. The network has become more and more controllable, flexible and programmable. The primary motivation for VeriFlow [7] is to create a tool that can check network invariants in real time, and one that is easily scalable, extensible and maintainable.

Many of the past related works have successfully check the network invariants, but none of them can do the work real-time, such as [2], [4], [6], but none of them meets the real time requirements from network administrators. VeriFlow is designed to be a real-time tool by introducing the concept of Equivalence Classes. In a nutshell, Equivalence Class represents an IP address range where packets that fall within the same EC would share the same forwarding behavior. Unlike other tools that would construct the entire forwarding graph of the network to check for invariants, the introduction of this concept would allow us to cut the network into many ECs and then check the network invariance only within affected ECs. This would theoretically allow us to reduce the computation complexity by a great amount. More details on this will be covered in.

VeriFlow is designed to run in an environment that has deployed SDN. More specifically, it serves as a layer that sits between the controller and switches. In an SDN environment, each switch is configured to talk to the controller, and so they are configured with the controller IP and port number before hand in order to connect to and then send/receive requests/responses from the controller. In an SDN environment that is configured to run VeriFlow, each switch is then configured to talk to the VeriFlow software, and so they are configured with the IP and port number the VeriFlow software is listening on before hand. From the switch perspective, it does not know whether it is connecting to the VeriFlow or the real controller because the settings for switches are the exactly the same in both configurations. So, there is no need to make any code changes to the software running on switch. In an SDN environment, the controller has connections directly to the switches. However, in an SDN environment where VeriFlow is set up, the controller is actually talking to the VeriFlow software. From the controller perspective, it does not know whether it is talking to the real switches or VeriFlow. VeriFlow thus contains 2N connections, where N is the number of switches, with N connections to the real switches and another N connections to the controller. When receiving a routing request from a switch, VeriFlow will just forward that traffic directly to the controller without intercepting it. However, when the VeriFlow receives a rule update packet from the controller, it will intercept the packet and feed it into the verifier, which will be briefed in the next several subsection. If the verifier does not flag an error to the rule update, VeriFlow will forward the update to its designated destination switch. Otherwise, the Veriflow software will drop the rule update packet directly.

2.2 Equivalence Classes

An Equivalence Class (EC) is a set P of packets such that for any p1, p2 belonging to P, and any network device D, the forwarding behavior is identical for both p1 and p2. In other words, a change in a forwarding rule R would only affect rules that would fall in the same Equivalence Class as rule R. All other rules would not be affected. Intuitively, this would save a tremendous amount of time and construct much smaller forwarding graphs instead of the need to construct the entire network forwarding graph, like most existing offline verification solutions.

For example, for one specific forwarding device R, there is originally only two forwarding rules specifying the destination IP in the network:

$$RuleA : 192.168.10.*$$

$$RuleB : 192.168.10.123$$

VeriFlow [7] will then generate three equivalence classes in this scenario that affects the equivalence classes of the following ranges:

EC_ID	StartIP (Inclusive)	EndIP (Inclusive)
EC1	192.168.10.0	192.168.10.122
EC2	192.168.10.123	192.168.10.123
EC3	192.168.10.124	192.168.10.255

Afterwards, a new rule C comes in with the following information:

$$RuleC : 192.168.10.128$$

This new rule C will not cut EC3 into three new Equivalence Classes, but will not affect EC1 and EC2. Here is the new formation of ECs:

where EC1 is affected by Rule A, EC2 by Rule A and Rule B, EC3 by Rule A, EC4 by Rule A and Rule

EC_ID	StartIP (Inclusive)	EndIP (Inclusive)
EC1	192.168.10.0	192.168.10.122
EC2	192.168.10.123	192.168.10.123
EC3	192.168.10.124	192.168.10.127
EC4	192.168.10.128	192.168.10.128
EC5	192.168.10.129	192.168.10.255

C, and EC5 by Rule A.

With the insertion of Rule C, only EC4 can be potentially affected, so we only need to construct the graph for EC4 to see if there are any problems in the network.

This is an over-simplification of the actual problem as it only involves the IP address match of one level. In the actual implementation, Veriflow traverses down the lookup data structure of several levels and try to find the as many Equivalence Classes as possible.

The algorithm Veriflow used to find Equivalence Class is rather complex and is tightly coupled with the choice of the proper look-up data structure. They will be covered in Section 2.5 in more details.

2.3 Single-bit Trie based lookup data structure

One of the single most critical piece of component that Veriflow [7] implements is a very efficient prefix lookup data structure tailored for IP addresses. The lookup data structure has to be extremely efficient for the following three operations:

1. Inserting a new rule, modifying or deleting an existing rule.
2. All rules described by a given prefix can be retrieved, much like a simplified version of regular expression match with $*$ involved.
3. Slicing an equivalence class can be done quickly.

There are many potential candidates for this data structure, as we will elaborate in later sections. But the most intuitive one, the one that is adopted by current Veriflow, is a multi-level single-bit multi-ary Trie based on IP address format. Trie is also named prefix tree, from which one can see it is efficient for prefix matching. It is a multi-ary tree, representing a prefix by a path in the tree starting from the root, where each node represents a bit. Each node has three branches, a zero branch, a one branch and a $*$ branch. Rules are stored at the leaves of the Trie. This data structure can be easily implemented and the three operations required by VeriFlow can be performed naturally by following the path of the each node given an IP address. Since each rule contains multiple levels of IP address fields, the Veriflow actually implements a multi-level Trie, where the leaves of intermediate levels actually contain the root to the next level Trie. The rules are then stored at the bottom level trie leaves.

We did some experiments about the efficiency of using this data structure, and found out that there can be better solutions, such as multi-bit trie, etc. The 32-bit nature of IP address and the vast number of fields in the packet can make the look up process slow and the Equivalence Class slicing process rather complicated and inefficient. More details on this will be covered in Section 2.5.

2.4 Graph Constructions and Graph Traversal Algorithm

The Veriflow [7] constructs a forwarding graph for each Equivalence Class constructed using algorithm and data structure proposed in the previous steps. Intuitively, if the new rule being inserted/deleted/modified is more specific, the number of affected Equivalence Classes should be smaller as there will be fewer affected rules. In the other hand, however, if the new rule changed is a very general rule, involving many $*$s, many of the existing Equivalence Classes will be affected, thus increasing the time for graph construction and graph traversal.

As is talked in previous sections, an over-simplified forwarding rule usually contains the following 2 major pieces of information that we care most: currentLoc and nextHop. In fact, we are only using these two pieces of information to construct the graph. Nodes in the graph represent a device, and edges in the graph represent rules.

Traversal in this graph in the original Veriflow implementation is done in the following way. Starting at the node that represents the currentLoc, traverse the graph by following the outgoing edges using Depth-First Search. Although there may be many outgoing edges at a single node, the traversal algorithm will only traverse the edge that has the highest priority. Thus, intuitively, the traversal of the algorithm takes very little time while the construction of the graph can be rather time-consuming.

2.5 Optimization objectives

After experimental analysis of the existing Veriflow [7] system, we found out that the system can be optimized in all of the above aspects.

2.5.1 Optimizing Equivalence Class

As described in Section 2.2, an Equivalence Class represents a set P of packets such that for any p1, p2 belonging to P, and any network device D, the forwarding behavior is identical for both p1 and p2. Given the example we talked about in Section 2.2, we showed that the insertion of a new Rule C would generate a total of five Equivalence Classes. However, if we look from a difference perspective, we can see that since EC1, EC3 and EC5 are all only affected by Rule A, the forwarding graphs constructed in those ECs are essentially the same. So, by following this intuition, we should be able to greatly reduce the size of ECs by eliminating near-half duplicate ECs, and thus reducing the verification time by half. Further, the original DFS-like algorithm for finding all Equivalence Classes on the multi-level Trie would not allow easy parallelization. So, a new algorithm to finding Equivalence Classes is also highly anticipated.

2.5.2 Optimizing the Lookup Data Structure

As described in Section 2.3, the original Veriflow uses a Trie data structure to store the rules and execute the algorithm for retrieving Equivalence Classes. However, as we figured out during the experiment, a Trie can be highly imbalanced, leading to a waste of space if the number of rules on a path is significantly smaller than the path length, or the number of nodes on the path. Also, due to the imbalance, a query operation does more pointer dereferences/memory accesses than necessary in the optimal case. This cache-unfriendly behavior can be a serious performance bottleneck for real-time tools like VeriFlow. Further, this data structure needs to be well suited for the new algorithm for finding non-duplicated Equivalence Classes. A new efficient data structure needs to be designed to meet this goal.

2.5.3 Optimizing Graph Construction and Traversal Algorithm

A bulk overhead of the original Veriflow is in its graph construction phase. In the original implementation, the algorithm would first find out all the Equivalence Classes, and then construct a list of graphs for each EC one by one. Afterwards, it will traverse all constructed graphs one by one. This can be very time-consuming and waste a lot of memory. Intuitively, one can optimize this phase in many aspects. For example, pipelining the graph construction and traversal would be a good idea. However, our focus is primarily on optimizing the core algorithm for graph construction that would both save time and space.

2.5.4 Other Optimizations

After carefully looking at the original Veriflow codebase, we felt that we can optimize the system in some further aspects, such as software engineering. For example, we found out that the original Strings representation of IP addresses are not optimal because C String operations are inherently slow. We also see the opportunity to fully parallelize the packet search phase by trying to design a BFS-like approach for finding Equivalence Classes in the Multi-Level data structure. Another example would be designing a caching scheme to cache past results because we found out that many rule updates are almost identical by analyzing the snapshot we used in our testing. Further, we used a tool called Intel V-tune [13] to benchmark the entire program and found out that the entire code structure can be much more optimized.

Chapter 3

Data-Structure and Algorithm Related Optimizations

The most important aspect of our work is that we have almost revamped the entire stack of core data structure and algorithm related to efficient adding, removing, or modifying rules, and finding the Equivalence Classes given a new rule. First, we defined a new representation for IP address that would vastly boost the speed by extensively using bitwise operations. Second, we spent a majority of time comparing many different possible alternate data structure options by surveying the literatures and doing experiments, especially for prefix-matching tasks. Next, we defined our new Restriction Class concept and designed our new algorithm for finding Restriction Classes based on top of that new data structure. Finally, we spend a huge amount of time simplifying the codebase organization, and debugging the code to make sure that everything is correct. This chapter will start with an introduction to our newly define concise IP address representation. Next, it will introduce the new data structure used to achieve fast packet search. Finally, the chapter will end with an introduction to our newly defined concept, Restriction Class, and the new algorithm for finding them.

3.1 Concise IP Address representation—IPPrefix

No matter what data structure we opt to use, an efficient IP address representation is always required to order and index the data structure properly. The original Veriflow [7] uses a String representation, but we felt that falls short of our requirement. In one extreme, for data structure like Trie, we need to access every bit in sequence in the representation of IP address to traverse the entire tree. Thus, a String representation would be cumbersome because we need to first convert the String into 16-bit representation. In the other extreme, for the data structures like Hash Table, we need to efficient generate the hashcode for the IP and compare the two IP address very quickly. Generating hashcode for String is slow and comparing them requires checking each character in the string. So, instead, we introduce a new data type called IPPrefix to store the packet field values. IPPrefix is a C++ pair containing two fields, a $uint64_t$ that stores the packet field value and a $uint8_t$ that stores the mask length. In $C++$ representation, The IPPrefix itself is of type $std :: pair < uint64_t, uint8_t >$. The first field of this pair indicates our representation of an IP. The second field of this pair indicates the length of the prefix. The following table contains a couple of sample mapping between IP address and the value of IPPrefix. Notice that the interesting bits start from the left most bits in the first field, the $unit64_t$ field.

IP Address	IPPrefix
192.168.0.0/16	$< 0xC0A8000000000000, 16 >$
192.168.24.0/24	$< 0xC0A8180000000000, 24 >$
192.168.24.25/0	$< 0xC0A8181900000000, 0 >$

The first field is a 64-bit number because in one of the levels, the address is actually 64 bits. The second field is only an 8-bit number because this field represents the mask length, and no mask length would exceed 255. Thus the size of this representation is always 9 bytes, always shorter than any kind of String representation. This IPPrefix supports the following operation very efficiently:

1. Accessing every or a group of bits from left to right

   This operation is required by many data-structures, especially for those trie-like data structures. Single-bit trie would iterate over each bit in the first field to traverse down the trie. A multi-bit trie would iterate over a group of bits in the first field down the trie. In both scenarios, a traversal is terminated by witnessing a $*$ bit or it will visit every bit in the first field. Since the first field is of $unit64_t$ type, iterating over each bit or a group of bits cannot be easier. For example, to iterate over each bit, we first do a bitwise AND operation with $0X8000000000000000$ to get the bit value and then shift left by one bit in each iteration. To iterate over a group of 4 bits, we first do a bitwise AND operation with $0XF000000000000000$ to get the bit value and then shift left by four bit in each iteration.

2. Generating hashcode

   The IPPrefix is composed of two elements, a $uint64_t$ and a $uint8_t$. Although the two fields are of different types, they can be casted back and forth. In our implementation, we first cast the second field from $uint8_t$ to $uint64_t$ so that those two fields can easily be manipulated. When deciding which algorithm to use, the first requirement for us is that it should be super fast, and second, it will not cause many hash collisions. We figured that using an XOR operation between the first field and the casted second field to generate the hash might just be efficient enough for us. First, it only uses one bitwise operation so it should be fast. Second, the values that we care about about in the first field tend to clutter together in the left and rarely do they occupy the entire 64 bit space while the values that we care about in the second field tend to clutter in the right, so hash collisions should not be common. We did some experiments against many different hash algorithms and ended up proving that we are right. The XOR is extremely time efficient and reasonably space efficient.

3. IPPrefix Comparison

   This functionality is required by data structures like hashtable when a hash collision happens. In a system where String is used to represent the IP address, comparing if they are the same is just calling a String comparison function. However, the overhead of calling a String comparison function is not minimal. In our implementation, we are just comparing if the two field values are the same as the other one. Since they are all of type $uint64_t$ and $uint8_t$, the comparison can be finished in just one bitwise operation. Actually, we also implemented a comparison function for IPPrefix so that the ordering of IPPrefix can expose the ordering of IP address. We will talk about it in detail when we introduce our new data structure.

3.2 Efficient Look-up Data Structure

As we talked in Section 2.5, one of the primary optimization objectives of the Veriflow [7] is to optimize the packet search by using a new kind of data structure that best satisfy the following requirement:

1. Inserting a new rule, modifying or deleting an existing rule.

2. All rules described by a given prefix can be retrieved, much like a simplified version of regular expression match with $*$ involved.

3. Slicing an equivalence class can be done quickly.

There are many data structures that can achieve the above three goals and there are a couple papers concerning the optimization and extensibility of those data structures in certain scenarios, such as DMP-Tree [12], Prefix Hash Tree [9], etc. We considered several core alternative options and finally settled down to use a Red-Black tree based solution. Some alternatives which we have taken into account will be briefed in the Related Work Chapter.

3.2.1 A height balanced tree based solution: Introduction

While there are many data structures that seem natural for the desired task, we found an alternative by using a height balanced tree based solution, which isn't talked in any papers. With some modifications, a height-balanced tree can be altered to meet the requirement of this task. Further, the vast existing libraries for height-balanced tree are so mature that we are certain that the internal implementation of the data structure is always optimal. Another advantage of using a tree-based solution is that since height-balanced tree libraries are so widely used in different tasks, there are also concurrent versions of this library. Many parties have proved the efficiency and reliability of those libraries. By leaving the majority of data structure internal implementation to the reliable third party, we can focus on getting the upper levels correct and fast.

3.2.2 A height balanced tree based solution: Implementation

Using C++ $std :: map$ implementation

In this case, we considerred that using the s standard librarys implementation of $std :: map$, which is implemented internally with a red-black tree , would carry more benefits. Including the benefits mentioned above, this decision is made primarily based on the following reasons:

1. Red-black tree is a type of balanced binary search tree, whose lookup time is only dependent on the total size of the tree. It does not have unnecessary internal tree nodes to keep the data structure organized they way trie does, leading to a more compact in memory storage.

2. The standard librarys robustness is guaranteed since it has been used and tested for much production software.

3. It could simplify our implementation, as the library has already provided us sufficient APIs to do the query operations we need.

4. Using this library relieves us from all low-level optimizations efforts associated with using that data structure.

Replacing Prefix Retrieval with Range Queries

The map C++ library is essentially view as a key-value store, so it provides the very basic operations such as insertion and removal in just one line of code. However, since it is internally implemented as a Red-Black tree, it requires an ordering for keys and thus provides many more features such as range queries. This range query feature is an essential piece in our new implementation, because it can provide the functionality to do prefix retrieval. Consider the following example:

ID:	IP Address	IPPrefix
A:	192.168.*.*	$< 0xC0A8000000000000, 16 >$
B:	192.168.24.*	$< 0xC0A8180000000000, 24 >$
C:	192.168.24.25	$< 0xC0A8181900000000, 0 >$
D:	192.168.25.*	$< 0xC0A8190000000000, 24 >$
E:	192.168.24.26	$< 0xC0A8181A00000000, 0 >$
F:	192.168.0.*	$< 0xC0A8000000000000, 24 >$
G:	192.*.*.*	$< 0xC000000000000000, 8 >$

Our intention is to convert prefix retrieval into a simple range query command. To achieve this, we wanted to order the IP Addresses in the following manner:

1. Replacing mask bits with zeroes, and compare their IP Address values directly. Ones with larger values are larger.

2. If two IP Addresses have the same value with * replaced with zeroes, then compare their length of mask. Ones with fewer mask bits are larger.

This might seem a little bit of unintuitive at beginning, but walking through an example would soon make sense. According to the above rules, the 4 mentioned cases would have the following ordering:

$$D > E > C > B > A > F > G$$

With this ordering rule in mind, we can do our trick to convert Prefix Retrieval into range queries. Given a new IP Address, our first task is to generate the lower bound and the upper bound for that IP Address. Typically, the lower bound is itself; the upper bound is the input with the field value incremented and mask value of 64. The mask length of 64 ensures all the existing field values matching the packet field are included without false positive.

For example, if given the IP Prefix 192.168.24.*, the lower bound value we generated is 192.168.24.0 and the upper bound value we generated is 192.168.24.255. Let us name 192.168.24.0 as X and 192.168.24.255 as Y. If putting these two new IP addresses into our original five IP address, we will then have the following ordering:

$$D > Y > E > C > B > X > A > F > G$$

We can see that address E, C and B will fall in the range, which is exactly we expected. Address E, C and B are addresses we would like to retrieve via 192.168.24.*

Consider another query with address 192.168.*.*. To run this query, we will compute its lower bound as 192.168.0.0, name it U and its upper bound as 192.168.255.255, name it V. Putting U and V into the comparison, we have the following ordering:

$$V > D > E > C > B > A > F > U > G$$

We can see that all rules except G will fall within this range. In fact, this is exactly what we wanted to expect. All of the rules from A to F contain 192.168 as the prefix.

As described above, the IPPrefix ordering is the trick to query a range of packet field values. From the implementation point of view, a comparison function is required to provide this feature. Firstly, the comparison function checks the 64 bits packet field value, because the IPPrefix with smaller field value is smaller. Secondly, the mask length determines the priority of IPPrefix if both IP Prefixes have the same field value. A smaller mask length means a smaller IPPrefix. With this comparison function, the IPPrefix ordering is ensured in our data structure. The Python style pseudo code is shown as follows:

```
def getIP(ipprefix):
        return ipprefix[0]


def getMask(ipprefix):
        return ipprefix[1]


def IPPrefixCompare(x,y):
        if getIP(x) == getIP(y):
                return getMask(x) < getMask(y)
        return getIP(x) < getIP(y)
```

Mutilevel Data Structure

Further, since a rule can contain many fields, we have to run the prefix match on each level of the field. Learnt from the previous work in Veriflow [7], which uses a multi-level trie to store all the rules, we borrowed the idea and used a similar multilevel structure to store rules. The only difference is that each level is actually a red-black tree instead of a trie. For all intermediate levels, instead of storing the actually set of rules, the leaf nodes contain a pointer referring to the next level map. We store all rules of the same packet header to the last level map leaves. We also did some field swapping to further optimize the data structure, read Chapter 4.1 for more details.

So far, we have covered the rationale behind choosing this data structure and introduced the trick to make it suitable in our use case. One last operation that we havent talked about, the one that is of the most importance, is slicing Equivalence Class in this multi-level tree structure. In the next section, we will talk about our new algorithm for slicing Equivalence Class.

3.3 Restriction Classes and a stack-based algorithm

In this section, we will first introduce the concept of Restriction Class, a term we use to denote Equivalence Classes. Next, we will talk about the stack-based algorithm to find Restriction Class which is able to find Restriction Classes in linear time at one level. This can then be divided into another two parts: 1: A new stack based algorithm to find Restriction Classes in one-level. 2: Extending the first algorithm and make it applicable into a multi-level balanced tree environment using a BFS based approach.

3.3.1 Restriction Class

As is mentioned in Chapter 2.2, an equivalence class represents a partition, multiple ranges of packets fields, of packets in which the forwarding behavior is only affected by the rules matching the field. In the process of verifying a new rule, traditional Veriflow [7] takes the new rule and the entire network forwarding rules affected by the new rule to generate equivalence classes. In the Chapter 2.2, we introduced the following example:

Rule A: 192.168.10.*

Rule B: 192.168.10.123

Rule C: 192.168.10.128

Given these three rules, we can generate the following Equivalence Classes according to the algorithm proposed in the original Veriflow:

EC_ID	StartIP (Inclusive)	EndIP (Inclusive)
EC1	192.168.10.0	192.168.10.122
EC2	192.168.10.123	192.168.10.123
EC3	192.168.10.124	192.168.10.127
EC4	192.168.10.128	192.168.10.128
EC5	192.168.10.129	192.168.10.255

However, if we think carefully about the concept of Equivalence Class, what we care most is not the Equivalence Range itself, but rather the set of rules that will affect a certain Equivalence Class. For a certain Equivalence Class, we need to construct the forwarding graph of that EC based on the set of rules that affect it. Thus, we introduced the idea of Restriction Class, which denotes those rules that affect a certain Equivalence Class. The key observation we have here is that on average, there are far less Restriction Classes than there are Equivalence Classes because many different Equivalence Classes might share the same Restriction Class. The only exception is when every Equivalence Range represents only one unique IP Address. However, this barely happens in real world. For example, if we take a closer look at the example above, we observe where EC1 is affected by Rule A, EC2 by Rule A and Rule B, EC3 by Rule A, EC4 by Rule A and Rule C, and EC5 by Rule A.

Thus, we can merge the equivalence classes affected by the same set of rules into a restriction class. In a restriction class, the forwarding behaviors of packets are affected by the same set of rules. In the example, there will be three restriction classes (RC1, RC2 and RC3) generated. RC1 is affected by Rule A, RC2 is affected by Rule A and Rule B while RC3 is affected by Rule A and Rule C.

Optimization is the main goal in our work. Building restriction classes instead will reduce the number of forwarding graphs. In the original design, the verification process involves constructing a forwarding graph for each of the ranges given the rules affected. So, the number of forwarding graphs constructed is $O(ECs)$. By introducing the concept of restriction class, we can effective reduce the number of forwarding graphs constructed to $O(RCs)$.

3.3.2 Stack-based algorithm to slicing Restriction Class

In this subsection, we will briefly talk about the process of finding restriction class in a single level. We will consider a very simple example first. For the sake of demonstrating the algorithm of finding restriction classes at each level, this example only involves exactly one field in its packet header.

The process of finding the complete restriction classes of a packet that involves multiple fields in its header will be discussed right afterwards.

Consider the example that contains the following four rules:

Rule A: $92. * . * .*$

Rule B: $92.8. * .*$

Rule C: $92.68. * .*$

Rule D: $92.8.10.*$

The process of finding the restriction classes for these four rules starts with generating the lower bound (inclusive) and upper bound (exclusive) address for each rule. In the above example, we have:

Rule ID	Lower Bound	Upper Bound
A	92.0.0.0	93.0.0.0
B	92.8.0.0	92.9.0.0
C	92.68.0.0	92.69.0.0
D	92.8.10.0	92.8.11.0

Refer to code below for the implementation of finding the lower and upper bounds:

```
def getUpper(ipprefix):
(ipprefix[0]+(1<<(64-ipprefix[1]))-1, 64)
```

From the above example, we can see that the order of those rules is preserved in its lower bound representation, while the order of those rules is reversed in its upper bound representation. Since the rules we retrieve out of the range query function are actually in order, we can use this property to efficiently build the sorted list of values consisting of lower bound and upper bound. After getting the lower bound and

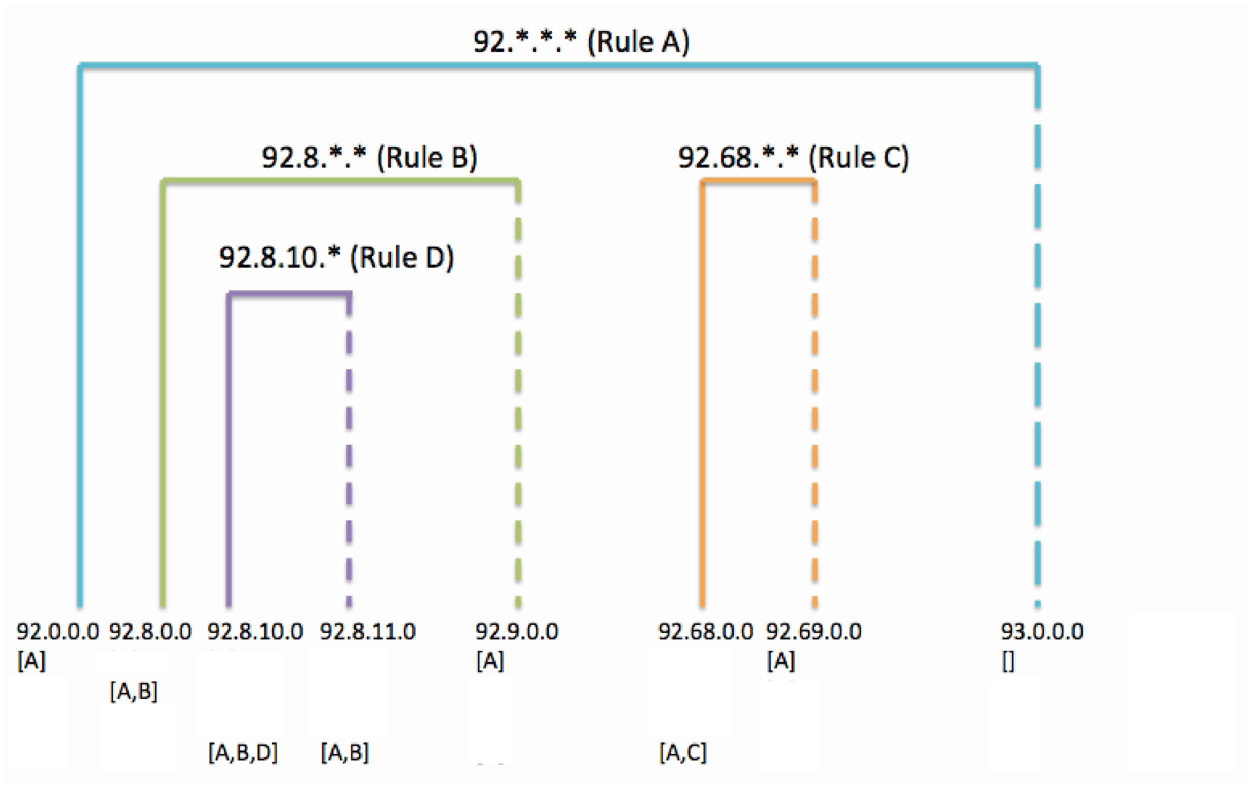

Figure 3.1: Example Stack

upper bound address for all of these rules, we put all the rules together and merge sort them according to the comparison function provided in IPPrefix subsection.

The result after sorting is something that looks very similar to a multi-level umbrella shape, see figure below. The spaces between each bound, whether lower or upper bound, represent each equivalence class. And as one can see from the figure below, there are always 2N+1 equivalence classes at each level with N rules given. After sorting is done, we then start the labeling process. The goal of the labeling process is to label each Equivalence Class properly so that it knows which rules will affect it. Since all the bounds are sorted, we will use a stack to mimic the labeling process so it will finish labeling in linear time. In 3.1, the values in side the bracket in each bar represents the current stack value at that moment.

There are many edge cases that make its implementation harder than thought, but the code below gives the overall idea:

```
Label(BoundList):
      S = Empty Stack
      RC = empty set
      For each bound value l in Bound List:
           if  l is lower bound:
                  S.push(l)
                  RC.add(S)
           if l is upper bound:
                  S.pop()
           Return RC
```

The 3.1 pretty much explains what is going on inside the stack. Notice that according to the code above, RC only adds current stack snapshot when the stack just finished pushing something into it. With that said, if we look back to the figure above, a snapshot would only takes place in the 1st, 2nd, 3rd and 6th iteration, making up a total of 4 Restriction Classes.

3.3.3 BFS based approach to do the packet search in all levels

In this subsection, we will briefly talk about the process of finding all restriction classes. In the above section, we have briefed the idea of finding the restriction class at each level. As is described in the previous subsection, each intermediate level tree node contains a pointer that points the next level red-black Tree. For each restriction class we found, we get all the tree nodes in that level that represents the rules in that restriction class. We then get all the next level rules of all the tree nodes found previously and then find their restriction classes. We do this in level order iteratively until we have reached the last level. Once we have reached the last level, we will get all the complete rule_set stored in the last level tree leaves and construct forwarding graph for each restriction class.

Here is the C++ Style Pseudo code:

```
vector<vector<Node*>  >* VeriFlow::getAllEquivalanceClasses(Rule& rule)
{
vector<Node*> initLevel;
initLevel.push_back(this->primaryTree.root);

vector<vector<Node*>  >* current = new vector<vector<Node*> >();
vector<vector<Node*>  >* final = new vector<vector<Node*>  >();
current->push_back(initLevel);

for (int i = 0; i < ALL_FIELD_INDEX_END_MARKER; i++) {
```

```
for(unsigned int j=0; j<current->size(); j++){
getAllRanges(rule, i, (*current)[j], final);
}
vector<vector<Node*> >* temp = current;
current->clear();
current = final;
final = temp;
}
delete final;
return current;
}
```

As you can possibly see, this pseudocode is just a modification of the BFS algorithm. Instead of just traversing down the tree, it also merges all the Restriction Classes down the path so that it returns all the Restriction Classes found after traversal finished. $vector < Node* >$ represents a Restriction Class. $getAllRanges()$ is a function we used to retrieve all the Restriction Classes in a single level. Two queues are used interchangeably in the above code, the current queue and the final queue. They respective represent nodes in the current level and next level.

Chapter 4

Software Engineering Related Optimizations

We split this chapter into two section: Optimizing Packet Searching and Optimizing Graph Construction. Graph traversal is not taken into account here because experiments have shown that the time to do graph traversal is almost negligible.

4.1 Optimizing Packet Searching

Although the new data structure and algorithm described in the previous Chaper significantly reduced packet searching time by 5 times, we can still find rooms to further increase that number. Actually, a non-negligible portion of time reduction in packe t search phase should be credited to the introduction of Restriction Class, which significantly reduces the overhead by merging same Equivalence Classes. We strongly desired to figure out something new.

4.1.1 IP field swapping

As we delve deeper into the code runtime optimization, we figured out that the multi-level nature of this data structure has significant impact on the runtime of slicing Restriction Classes. Theoretically, there can be a huge number of Restriction Classes given that there are 14 levels. However, the actual number of Restriction Classes does not reflect the fact that there are 14 levels. After looking at the real world data set, we found that 13 out of the 14 fields in the packet are $*.*.*.*$ rules, making a vast majority efforts of slicing Restriction Classes in those levels meaningless. The only field that actually contains values is the 9th field that represents destination IP address. However, keeping the only useful field as the 9th field is not optimal at all. In fact, putting the IP Address in earlier field and thus upper level in the multi-level tree would only increase the overhead of slicing Restriction Classes. This is because we have to go through many repetitive and unnecessary processes down the multi-level tree of generating Restriction Classes that is trivial. Remember that we talk about the BFS-like approach to traverse down the tree and construct the Restriction Classes. At each stage, we are merging Restriction Classes. Imagine that we are only merging those in the last level could save us a lot of efforts, both in space and time. The optimal solution we found is to put it in the last level. This optimization boosts the speed by reducing 1/3 of the packet search time.

4.2 Optimizing Graph Construction

At this stage, graph construction would still take more than two thirds of the total verification time, 2.5 times as much as packet search phase. Actually, graph construction has always been the most time-consuming tasks in the verification process. In the original implementation, graph construction takes even more than four times longer to finding Equivalence Classes and takes about 80

4.2.1 Base Graph Construction

The intuition behind this optimization is very simple. Before this optimization, we will construct as many forwarding graphs as there are Restriction Classes one by one, and we will then traverse each constructed forwarding graph to test if there is any issues with the network. Since the bulk remains on graph construction instead of graph traversal, we are thinking if we can reuse any component when building each forwarding graph. With Base Graph, we will first look over all graphs we are going to build and trying to find the most commonly shared portion of graph and build that portion up first. We can always reuse that base graph for future forwarding graph construction, thus potentially saving a lot of time. The implementation of this is harder than thought though, because the tricky issue here is to choose to what degree should the base graph contain. Of course we want the base graph to contain as more nodes and edges as possible, but that also increases the overhead of finding the base graph, and might also not catch all the graphs. In the other extreme, if we only want that graph to contain the common information all graphs have, then it might not be so worth doing so given the overhead to construct it as this base graph tend to be small sometimes. Sometimes, some very large portion of the Restriction Class shared a huge graph, while the other few rules does not have any intersections with the rest. This can make the approach sitting on the other extreme side worthless. We are actually traversing all the graph to find their common components. In the end, we found out a solution that would only dive into three levels of depth.

4.2.2 Merging Graph Construction into Rule updates

Solving every engineering problems comes with tradeoffs, and it is the same with this one. We are trading some efficiencty in packet search and graph traversal for more efficiency in graph construction. To further reduce the time in the graph construction phase, we considered merging graph construction with rule insertion/deletion and directly storing the graph at the leaf node in the very last level. The rationale behind this is that we found that the representation of a graph is essentially a hash table, where the key in each entry is the currentHop information and the values in it are lists of their nextHop destinations. We also found that we will always need to store the Rules information in the leaf node in the very last level of the tree in a hash set. That means the operation to construct the graph in another pass is completely a waste of time. So, we think it is completely possible to merge these two processes and make graph construction an integral part of rule insertion/removal. For each new rule update, we update the graph by manipulating with the C++ unsorted_set. The time to do this operation is comparable with the operation of adding a new Rule into a set. The result of this merge is phenomenal, the graph construction phase is collapsed and the rule insertion/removal cost barely increases much. The only requirement is a new traversal algorithm that would concurrently traverse multiple smaller graphs because a graph is represented by a set of nodes that fall within the same Restriction Class. While many might think this is a huge overhead, it is actually not because the graph traversal time is so minimal that any significant increase in it will not have a huge impact on the overall result. In the next subsection, we will talk about the multi-graph traversal algorithm.

4.2.3 Multiple-graph traversal algorithm

The original Veriflow [7] bases its traversal algorithm on a DFS algorithm. Although it is traversing in a DFS fashion and the graph might be large, it actually traverses very few edges. This is because the traversal algorithm Veriflow adopts will only traverse edges that have the highest priority. This aspect actually makes the graph traversal a very trivial component in terms of optimization. We use the following algorithm to traverse multiple graphs concurrently:

```
\resizebox{\textwidth}{!}{%
l = list of hashtable


def traverse(currentHop):
        if(currentHop is visited)
               return;
         mark currentHop as visited.
         currPriority = LOWEST
         nextHop = NULL;
         for each h in l:
               if h[currentHop] not exist:
                continue;
            else
                if  h[currentHop].getHighestPriority().priority
                                > currPriority
                nextHop = h[currentHop].getHighestPriority()
        if nextHop != NULL
        traverse(nextHop)
```

As one can expect, this algorithm can get very slow if the degree in each node is high, and when the number of hashtable becomes large. But this is never a problem in our case, because we are always traversing the edge that is of the highest priority and that means the degree in each node is always 1.

Chapter 5

Evaluation of Optimizations

In this chapter, we will prove that our optimization works and works correctly. Proving it works means we are seeing significant increase in verification speed. Proving it works correctly means that we can produced the exact same prediction as the original Veriflow [7]. In the first section, we will introduce the setup of our experiment, including introducing the data and metrics we used to benchmark of optimizations. In the second section, we will show you the analysis of our experiment results we retrieved from running dataset described in the first section using metrics proposed there.

5.1 Experiment Setup

We modified VeriFlow to implement all the previously mentioned optimizations. This experiment is conducted in two scenarios, theoretic experiment and real-world workload experiment. We did performance evaluation in both conditions and proved that our optimizations have achieved significant results.

5.1.1 Machine Setup

All experiments are conducted on machine with the following specification: Quadcore Intel Xeon CPU 2.27GHz, 8 GB of RAM, 2 MB of L1 cache.

5.1.2 Theoretical Dataset Setup

A random graph generator is written using Python package networkx, to provide a network topology for emulating real-world network flows. It is configurable with number of middle boxes in network topology and a random seed number.

Specifically, we assign a unique IP to each node in the generated graph and randomly pick a pair of source and destination nodes. Then we do a path finding for the pair and generate the forwarding rules for this source-destination flow. The packet headers are the same for all rules in a flow. A rule also contains a local device IP and the next hop IP to indicate the forwarding direction.

1. Simple Unique Rules

   In this experiment, we would expect a short rule verification time, because the uniqueness (not subnetting) of IP results in zero number of equivalence classes affected by the new rule. We set up experiments taking an input number n (from 4 nodes to 799 nodes) as the number of nodes in generated graph and output the running time of each phase. Note that for the sake of the experiment, the number of routes created is proportional to the number of nodes in the random graph

2. Subnet Forwarding Rules

   Subnetting occurs commonly in real world forwarding rules. For example, all packets matching a destination of IP 192.168.0.0/16 should be forwarded out of router R over link L. For this experiment, we enhance the rules generator of our testing script to support subnet-forwarding rules. For each inserted rules, we randomly set the mask bits length from 2 to 32 in IP field, and the first bits in all IP are always one. Inside each rule verification, we send query of 128.0.0.0/1. This generates more equivalence classes than the previous test, as more existing rules are affected

5.1.3 Real world dataset

In this experiment, we simulated a network consisting of 172 routers following a Rocketfuel [10] topology—AS 1755, and replayed BGP RIB and update traces collected from the Route Views Project [1]. A BGP RIB snapshot consisting of 5 million entries was used to initialize the network FIB tables. Then we replayed a BGP update trace containing 90000 updates to trigger dynamic changes in network. This is the exact same experiment setup as the one described in the original Veriflow [7] paper.

5.1.4 Evaluation Metrics

Checking Correctness

There are many ways to check for the correctness of our system against the original Veriflow System, and they lie in different levels of abstractions. For example, one can check if the two algorithm have found the same set of Equivalence Classes. Another example would be to test if the $verify()$ function in two implementations are returning the same code, meaning that they have detected the same error or they both passed. In our case, both would likely to work, though we might want to collapse identical Equivalence Classes in the first method. The abstract final graph constructed in the new implementation shall look the same in both implementation given that their found ECes or RCes are the same. However, we choose to use the second method as that looks more flexible and viable in our scnenario. Comparing each set of found

Equivalence Classes or Restriction Classes is way slower than just looking at the return code. Further, since we have our own implementation of graph traversal, it is crucial that we put that into account as well. Obviously, the first method would skip that part. In fact, a large portion of the original concept are not taken into account so that it is always best for us to test for correctness at the highest level, although that will increase the effort to narrow down the error if there are any inconsistencies. So, instead of just printing out statements when an error occur, we actually returns a value back which indicated the value. When we run the actual data on both implementations, we store the errors we detected in a file and they diff the file afterwards. Implementation wise, Tthere are other challenges too. For example, a loop and a blackhole may happen at the same time in a graph. So, instead of of just return an int code, we are actually returning a set of integers.

Benchmarking Performance

Improving the performance is the primary goal of this work, so carefully defining the metrics to benchmark performance is important. Like the original Veriflow, we divide the verification process into the following steps:

1. Packet Search Time: This includes inserting a new rule into the data structure, and finding all the Restriction Classes given the insertion of that rule
2. Graph Construction Time: This includes traversing all found Restriction Classes, building a forwarding graph for each of the Restriction Class, and then store the set of forwarding graphs into a vector
3. Graph Traversal and Metadata Cleanup Time: This includes traversing all forwarding graphs stored in the vector one by one, cleaning up all dynamically allocated memories and construct the result back.
4. Total Verification Time: As the title suggests, this includes all the above mentioned times.

In this study, we are interested in the relationship between the average value from these metrics and the optimizations we have given the two type of dataset we described earlier in this chapter. We are also interested in the fluctuation of these metrics when giving the two different type of dataset to better understand our performance.

5.2 Result Analysis

5.2.1 Correctness Test

We saw a near 100

5.2.2 Performance Evaluation

Theoretic Experiment

Theoretic Experiment may not best capture the implementation's performance in real world. Tests are deliberately designed to show that our implementation is faster in some extreme cases. So, only simple comparisons are performed in this dataset.

1. Simple Unique Rules The running time betweem the two implementations varies greatly when the number of nodes increases. In fact, the ratio of the old implementation$'$ running time vs the new implementation$'$s grow exponentially up to 10^7x when the number of graph nodes grow to 800. It is conspicuous that for the comparison of those metrices, our implementation outruns VeriFlow significantly. One of the primarily reason is that we have chosen the test cases carefully to show the significant reduction in Restriction Classes in the extreme scenario. In most cases, our implementation reduces the running time to less than 5ms in the above experiment.

2. Subnetting Rules
   In this experiment, we figure out that the speedup of total verification time will grow linearly when the number of nodes increases, while the ratio is relatively large. Figure 5-1 shows this. In the most complex scenario with 799-hundred graph nodes and 32477 forwarding rules, our implementation finishes verification within three milliseconds while it takes VeriFlow 59.862 seconds to complete. Figure 5-2 represents the ratio of the number of equivalence classes found in VeriFlow to the number of restriction classes in new VeriFlow given this dataset. When number of nodes increases, the ratio will increase such that we expect our implementation generates less restriction classes and the graph building time will reduce because we have less graphs.

   Real-world Workload Test

   While the above test might be carefully designed to show the advantage of Restriction Classes, the real world work load test is surely more persuasive. The increase in speed is more reasonable. In a nutshell, you could find the Table 5.1 showing the increase in speed with each optimization. Moreover, our implementation is significant more efficient in terms of space utilization. During the experiment, the original Veriflow [7] would use up to 8GB of memory while the memory usage of our implementation never exceeds 2GB. Figure 5-3 and Figure 5-4 show two perfect graphs of how much increase each optimization has visually.

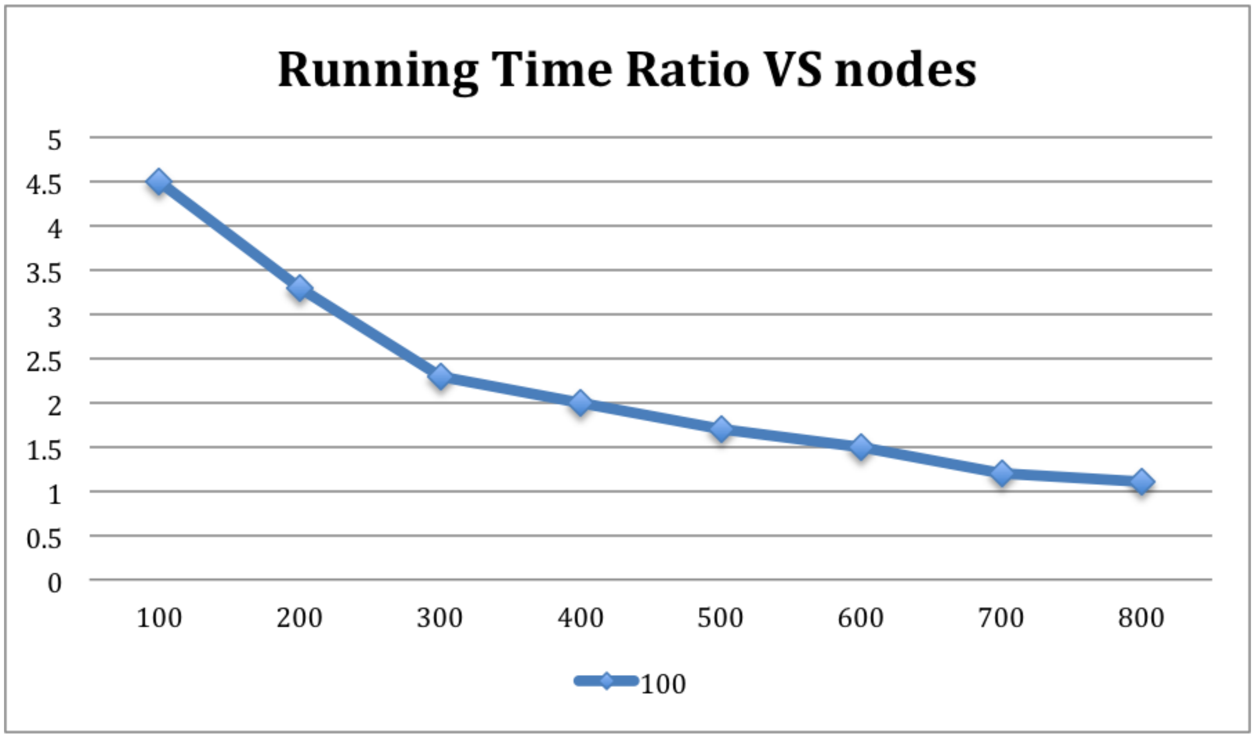

Figure 5.1: Running Time Ratio VS Number of Nodes

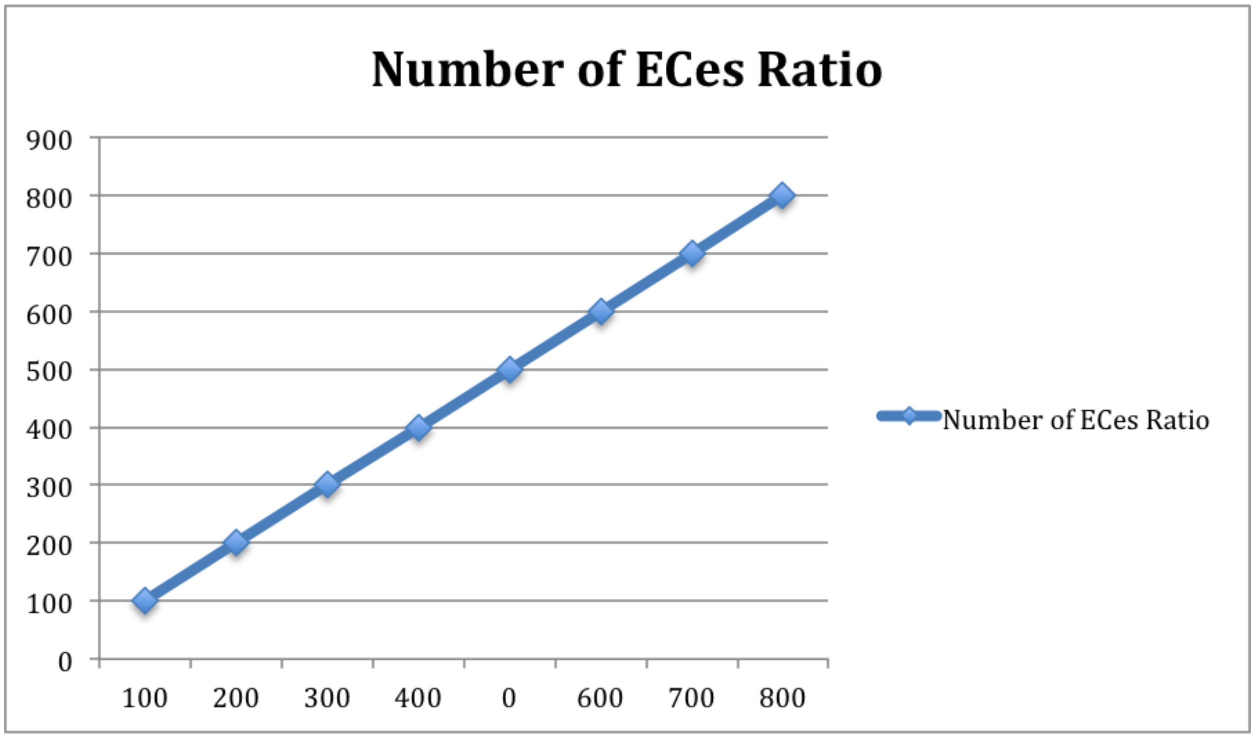

Figure 5.2: Number of ECes Ratio VS Number of Nodes

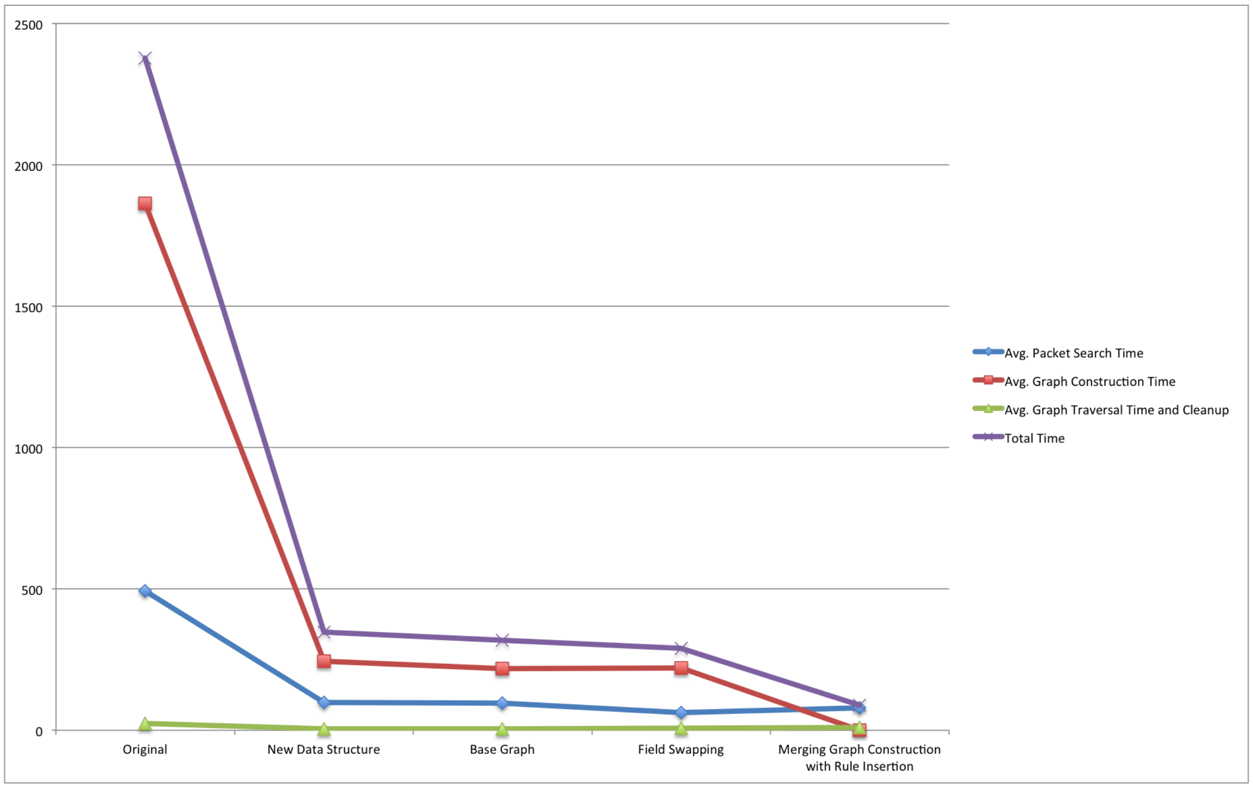

Figure 5.3: Running Time VS Each Optimization

Optimization	Avg. Packet Search Time	Avg. Graph Built Time	Avg. Graph Traversal Time & Cleanup	Avg. total time
Original	492.64	1863.3	23.44	2379.38
New Data Structure	98.65	244.79	4.52	347.96
Base Graph	96.24	217.22		313.46
Field Swapping	62.2	220.36		282.56
Merging graph build with rule insertion	78.6	0	9.56	88.16

Table 5.1: Performance Chart

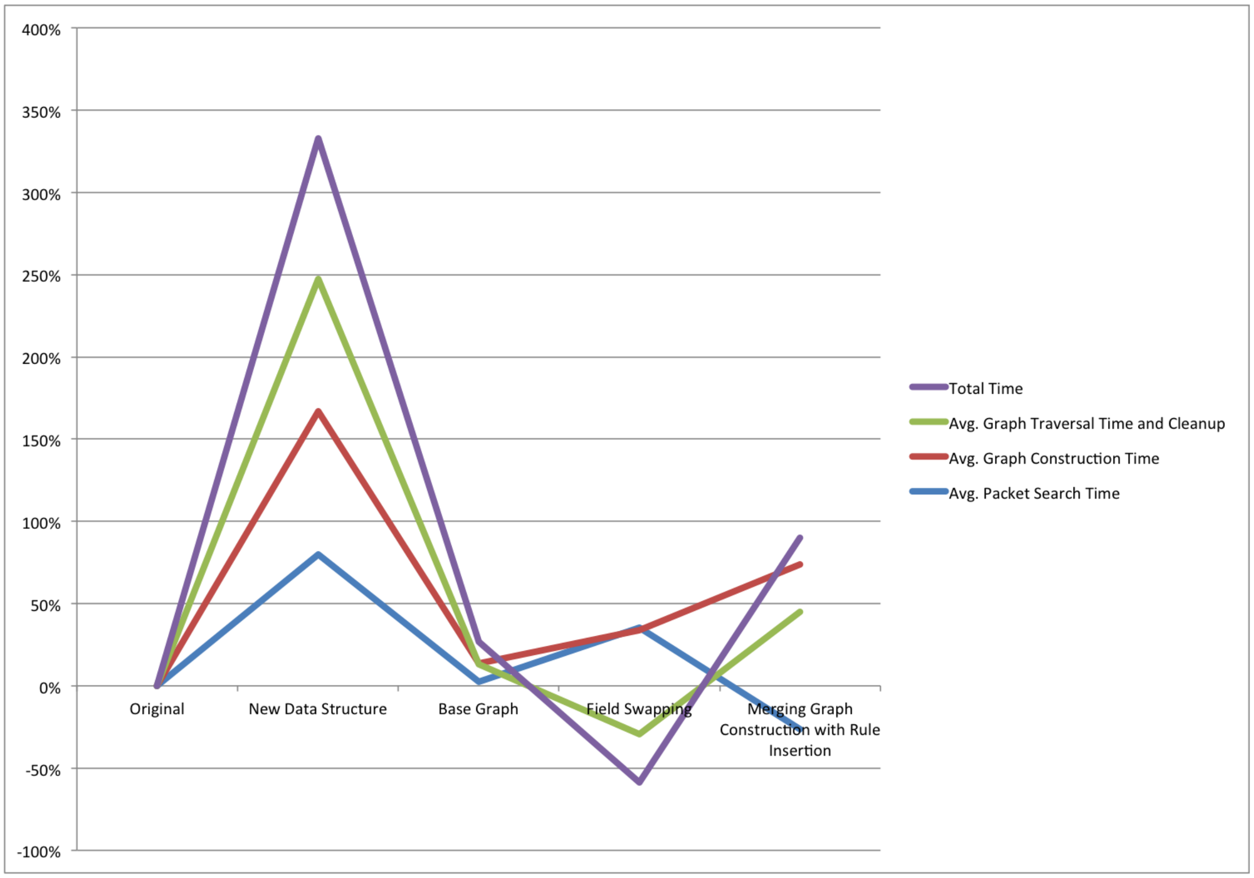

Figure 5.4: Running Time Increase Percentage VS Each Optimization

Chapter 6

Related Work

In this work, we proposed several optimizations. One of the vital piece of optimization is that we introduced a new kind of data structure to store the rules. There are many related works in this area, especially those on data structures which support prefix matching and non-algorithmic approaches that would boost prefix-matching performance. Each of them has their own pros and cons.

6.1 Caching

One of the first things one could think about when talking about optimizing performance is caching. Caching in network packet forwarding is talked in [20] by maintaining a mapper from a 32-bit value to another 32-bit value. [19] is an example of using caching in switches. In our usecase, this translates into maintaining an extra layer of cache inside the Red-Black tree on each level. This cache should maintain a mapping from an IP Address representation to two nodes, the one node which represents the lower bound and another which represents the upper bound. While caching can have a very high hit-rate in packet forwarding if backbone network shows a highly degree of locality, it does not work prefectly in our case primarily for the following reasons:

1. Data size concern

   Given that we maintain a multi-level RB-tree of more than 10 level, we can potentially create several thousand caches covering all branches, where each cache can contain up to 2^16 entries just for 16 bit address.

2. Node removal complexity

   Although adding data into the cache is easy and can be done in constant time, removing rules from the data structure can cause pain in manipulating the cache. We either want to sacrifice running time when we remove rules by checking each prefix entry possibility that may contain itself as the value, or we can sacrifice space by maintaining another mapping from itself to the prefix entry that contains it.

6.2 Single-Bit Trie

This is the data structure that is originally used in the Veriflow [7]. Trie is also named prefix tree, from which one can see it is efficient for prefix matching. It is a multi-ary tree, representing a prefix by a path in the tree starting from the root. It can be easily implemented and the two queries required by VeriFlow can be performed naturally:

1. all rules described by a given prefix can be get simply by following the path of the prefix and take all rules in the subtree whose root is the destination of path.

2. slicing an equivalence class is as simple as inserting a new subtree under the lowest common ancestor of existing rules and the new rule.

One thing good to notice about this data structure is that multiple queries can be run on the same trie in parallel easily. The implementation of a Concurrent Binary Search Tree has some implications for the parallel version of single bit trie mentioned above. Each node in the trie will contain a reader/writer lock and the reader lock will be grabbed for intermediate node if the action is an add/remove. The writer lock, however, will lock the leaf node. If any two instances of either kinds of operations do not involve conflicting read/write to the same rule, then those two actions are independent and can run independently without affecting each other. However, real world experiments show that the trie-based implementation does not scale good enough when the data size becomes large and when the IP addresses become sparse (hardly do they share most prefixes). A new data structure is highly desirable.

6.3 Pure Hash Table/Prefix Hash Tree

On the other extremely, a pure hash table implementation is also taken into our consideration. We are treating a hash table as a prefix tree by adding lots of upper layer logic on top of the hash table. One of the considerations of using a hash table based implementation is that it is supposed to add/remove entries very quickly without regard to the entry value itself, and these operations do take a long time in single-bit trie. We believe that this is a clear advantage against using a single-bit trie based solution. In a pure hash table based implementation, the efficiency to do action 1) —Inserting a new rule, modifying or deleting an existing rule, primarily derives from the time/space efficiency for hashcode. As is talked in Section 4.1, with the introduction of IPPrefix, hashcode generation for IPPrefix takes only few bitwise operations, and this is also true for IPPrefix comparison. Good hashcode generation reduces hash collisions, and fast comparison functions prepare the hash table for the worst condition. However, the pure hash table based solution is not perfect in terms of space efficiency. It is not a natural fit for doing prefix matches. In order to efficiently run prefix retrieval on hash table, we need to insert all its possible prefixes into the hash table. There are usually two ways to do prefix matches [27].

1. Binary Search on Ranges

   This is an intuitive way to do search in hashtable. In binary search on ranges, each prefix is represented as a range, using the start and end of the range. The algorithm uses binary search to find the interval in which a destination address lies.

2. Binary Search on Prefix Lengths

   Binary search must start at themedianpreïňĄx length, and each hash must divide the possible preïňĄx lengths in half. A hash search gives only two values: found and not found. If a match is found at length m, then lengths strictly greater than m must be searched for a longer match. Correspondingly, if no match is found, search must continue among preïňĄxes of lengths strictly less than m. Thus, it can achieve finding the match in $O(log2N)$ time, where N is the length of the prefix.

In a nutshell, the biggest advantage of using a complete hash table based solution is that it is very time efficient, even with running prefix match queries. However, it can use a lot of space in the worst-case scenario. The operation to slice Equivalence Classes is highly dependent on the efficiency to do prefix match queries. Thus, this solution can slice Equivalence Classes pretty quickly. One other very significant advantage of using this data structure is that it is extremely easy to scale to multiple machines. Prefix hash tree is a trie built on top of distributed hash tables, or DHTs [11] with logarithmic time lookup and double logarithmic time update. The idea behind it is similar to a pure hash table based solution. It supports prefix matching as well as range queries. Prefix hash tree can be considered as a potential data structure if VeriFlow [7] becomes a distributed system as it considers failure resistance and load balance. For this project, we focus on efficient query operation on a single host, and use prefix hash tree as enlightenment for high performance range query operation. If space is not a problem here, but scalability is, this solution might be a good fit.

6.4 Multi-level Hash Table/Multi-bit Trie

One of the data structures we considered implementing is multi-level hash table [3], which can be viewed as a generalization of Trie, or more specifically, a multi-bit trie. It is a choice between the two extremes of an either purely flat hash table or a single-bit prefix tree. The central idea of multi-level hash table is very similar to that of page tables used in virtual to physical memory address mapping: avoid the waste of memory space by constructing multiple levels of indirection and only reserve space for valid entries in each level. In our case, a flat hash table mentioned previously is just like a purely flat one level page table in memory mapping case. Specifically, multi-level hash table breaks an input key into multiple keys used for internal lookup, which we will call as internal key. At each level, an internal key is used to retrieve a pointer to the next level pointer, until the object is found in the bottom level hash table. It can be seen that if all internal keys are a single bit, multilevel hash table is essentially a trie. The main improvement of multi-level hash table over trie′s is the flexible definition of number of indirections, as well as the size of each internal hash table. It inherits most of trie advantage on supporting VeriFlow [7] queries, including embarrassingly parallel query processing. For range queries, multi-level hash table does at most L parallel lookups, where L is the number of masked bits for the internal key. However, unfortunately it also inherits tries drawback of requiring intermediate hash tables to maintain structure. But this is a decent choice between the two extremes.

6.5 Other Variation of Tries

There are all kinds of other variations of Trie data struction. A Level-Compressed Trie [21] is a trie in which every node contains no empty entries. It is a variable-stride trie. The motivation for this trie is to minimize storage by avoiding empty array elements. [24] is an example system that uses this data structure. It is very tunable, allowing faster speed in sacrifice of memory. Lulea-Compressed Tries [22] is a multi-bit trie which uses fixed-stride scheme. Central to this data structure is its use of bitmap. Bitmap compression is used to vastly reduce the storage size in each entry. [26] is a practical system that uses this data structure. Another kind of Trie uses Tree Bitmap [25] to achieve fast insertions in addition to what Lulea-Compressed Trie provides. Two bitmaps are stored in each trie node, one for internally stored prefixes while the other serves as the pointer which points to somewhere externally.

Chapter 7

Conclusion

Networks are complex and errors are common. Network administrators have always desired an efficient real-time networking debugging tool to check for network status. Traditional solutions, such as symbolic execution [13] tools like Klee [17], do not work well here due to the size of the web. They either run entirely offline, being too slow, or just cannot provide the adequate functionalities one would want. Veriflow [7] is a solution to it. Its goal is to provide network administrators with a complete set of network-monitoring utlities without sacrificing any speed. It does so by reducing the problem space from the entire network space into some small subsets of it, where each subset is called an Equivalence Class. Each Equivlancen Class represents a partition of network such that packets fall within this partition would have the same forwarding behavior. The key to the Veriflow success is the ability to satisfactorily complete a verification query in a large and complex real-world workload network. In this work, our goal is to further reduce the average and worst case rule verification time through optimizing querying data structures, simplying Equivalence Class searching algorithm, etc. More specifically, this boils down to the following:

1. Adding, Removing, and Modifying Rules in an instant
2. Finding Equivalence Classes faster
3. Reducing the number of Equivalence Classes
4. Optimizing the algorithm to generate and traverse the forwarding graph within an Equivalence Class

Some of the major observations and optimizations we made to address the points listed above:

1. A new notion of Equivalence Class and a new algorithm to find them

   By introducing Restriction Class concept to replace Equivalence Class in implementation level, we ended up designing a new algorithm to find Restriction Classes. This new algorithm leads to a faster querying speed, and a huge reduction in the number of final Equivalence Classes.

2. A new data representation and querying data structure

   By designing a new data representation and querying data structure, we simplied the procedures to add, remove and modify rules to just one line of code. We also opened opportunities to a variety of ways of parallelizing the packet search phase that we will talk about in Future Work below. Combined with the new Equivalence Class Finding algorithm above, we significant reduced the time to finding all the Equivalence Classes.

3. Simplying Graph Generation and a new algorithm for Graph Traversal
   Graph generation time has always been a major cost in verification time. By combining graph generation with rule insertion and proposing a new graph traversal algorithm, we significantly reduced the time to do graph generation and graph traversal combined.

With the new implementation and optimizations, we reduced the average total verification running time to 1/30 of the original implementation on real-world workload data, and reduced the memory usage to one fourth of the original implementation. As we felt that there are more optimizations can be done, we summarize a few of them in the Future Work Section below.

7.1 Future Work

7.1.1 Paralleling Packet Search

Packet Search phase can be devided primarily into two parts:

1. Finding matched packets in a single level
2. Applying the above algorithm and extend it to make it work in a multi-level tree

While we cannot think of more ideas to further bring down the time for finding matched packets in a single level, we think of ways to further reduce the time in the second part listed above. Since we used a BFS-based, or level-order traversal, approach to traverse down the multi-level tree, we can easily parallelize the algorithm we used for the second part. There is no complex dependencies between different branches in this multi-level tree. Actually, there have already been quite a few work on algorithm related to parallel BFS. [14] and [18] illustrate a method for doing parallel BFS. Further, we can overlay the process of packet search with rule insertion, and removal given that we can lock the data structure properly. [15], [16] are both papers which talk about this issue.

7.1.2 Pipelining Graph Traversal Phase

In addition to paralleling packet search as described above, we can also boost the overall speed of Veriflow by pipelining the graph traversal phase. Currently, graph traversal is done iteratively after the program finishes finding and slicing all the Equivalence Classes. The fact that all the Equivalence Classes have to be created before actually traversal takes place creates a barrier to performance. In the above subsection, we talked about parallelizing packet search, which is essentially finding all Equivalence Classes. Parallelizing that process means different Equivalence Classes are found at different times by different threads. This reveals an opportunity for optimization to us. Threads that finish their works early can start constructing graphs and traversing graphs early to reduce the overhead.

Bibliography

[1] University of Oregon Route Views Project.

[2] Ehab Al-Shaer and Saeed Al-Haj. Flowchecker: Configuration analysis and verification of federated openflow infrastructures. In Proceedings of the 3rd ACM workshop on Assurable and usable security configuration, pages 37–44. ACM, 2010.

[3] Myung Geol Choi, Eunjung Ju, Jung-Woo Chang, Jehee Lee, and Young J Kim. Linkless octree using multi-level perfect hashing. In Computer Graphics Forum, volume 28, pages 1773–1780. Wiley Online Library, 2009.

[4] Nick Feamster and Hari Balakrishnan. Detecting bgp configuration faults with static analysis. In Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2, pages 43–56. USENIX Association, 2005.

[5] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. In Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM, pages 3–14. ACM, 2013.

[6] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In NSDI, pages 113–126, 2012.

[7] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P Godfrey. Veriflow: Verifying network-wide invariants in real time. ACM SIGCOMM Computer Communication Review, 42(4):467–472, 2012.

[8] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. ACM SIGCOMM Computer Communication Review, 38(2):69–74, 2008.

[9] Sriram Ramabhadran, Sylvia Ratnasamy, Joseph M Hellerstein, and Scott Shenker. Prefix hash tree: An indexing data structure over distributed hash tables. In Proceedings of the 23rd ACM symposium on principles of distributed computing, volume 37, 2004. 53

[10] Neil Spring, Ratul Mahajan, David Wetherall, and Thomas Anderson. Measuring isp topologies with rocketfuel. Networking, IEEE/ACM Transactions on, 12(1):2–16, 2004.

[11] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. ACM SIGCOMM Computer Communication Review, 31(4):149–160, 2001.

[12] Nasser Yazdani and Hossein Mohammadi. Dmp-tree: A dynamic m-way prefix tree data structure for strings matching. Computers & Electrical Engineering, 36(5):818–834, 2010.

[13] Softwaretechnologie, A. K., Bernhard Aichernig, and Andreas Bauer. "Symbolic Execution and Program Testing."

[14] Leiserson, Charles E., and Tao B. Schardl. "A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers)."Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures. ACM, 2010.

[15] Lehman, Philip L. "Efficient locking for concurrent operations on B-trees." ACM Transactions on Database Systems (TODS) 6.4 (1981): 650-670.

[16] Bronson, Nathan G., et al. "A practical concurrent binary search tree." ACM Sigplan Notices. Vol. 45. No. 5. ACM, 2010.

[17] Cadar, Cristian, Daniel Dunbar, and Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs."OSDI. Vol. 8. 2008.

[18] Hong, Sungpack, Tayo Oguntebi, and Kunle Olukotun. "Efficient parallel graph exploration on multi-core CPU and GPU." Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on. IEEE, 2011.

[19] Myers Jr, Roy T., and Laurence N. Wakeman. "Forwarding database cache for integrated switch controller." U.S. Patent No. 5,740,175. 14 Apr. 1998.

[20] Varghese, George. Network algorithmics. Chapman & Hall/CRC, 2010. Chapter 11.3.1

[21] Nilsson, Stefan, and Matti Tikkanen. "Implementing a dynamic compressed trie."Algorithm Engineering. 1998.

[22] Varghese, George. Network algorithmics. Chapman & Hall/CRC, 2010. Chapter 11.7

[23] Sikka, Sandeep, and George Varghese. "Memory-efficient state lookups with fast updates." ACM SIGCOMM Computer Communication Review. Vol. 30. No. 4. ACM, 2000.

[24] Sahni, Sartaj, and Kun Suk Kim. "Efficient construction of variable-stride multibit tries for IP lookup." Applications and the Internet, 2002.(SAINT 2002). Proceedings. 2002 Symposium on. IEEE, 2002.

[25] Eatherton, Will, George Varghese, and Zubin Dittia. "Tree bitmap: hardware/software IP lookups with incremental updates." ACM SIGCOMM Computer Communication Review 34.2 (2004): 97-122.

[26] Bando, Masanori, and H. Jonathan Chao. "Flashtrie: hash-based prefix-compressed trie for IP route lookup beyond 100Gbps." INFOCOM, 2010 Proceedings IEEE. IEEE, 2010.

[27] Varghese, George. Network algorithmics. Chapman & Hall/CRC, 2010. Chapter 11.8 and Chapter 11.9