Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Roman May

Supercharging IP router memory with SDN

Semester Project
April to October 2015

Supervisor/Tutor: Prof. Dr. Laurent Vanbever
Co-Tutor: Thomas Holterbach

Abstract

The Internet has grown drastically in the last decades. As a consequence today's edge routers need to be capable of storing more than 500k routes [1] to cover all prefixes. This is a mandatory feature for each edge-router, even though most of the routes are almost never used, as N. Sarrar et al. in [2] show.

This project uses Software Defined Networking (SDN) to increase, or supercharge, a routers FIB size by adding a SDN-enabled software switch. We present an algorithm, which splits the FIB and only stores a part of it in the router's FIB, while the remaining routes are added as flow rules to the software switch. Additionally, the algorithm uses real-time traffic statistics to reallocate the routes, so that the most used routes are stored in the router's FIB, and only the less used routes are in the software switch.

The simulation results show that the proposed algorithm is linear to the number of routes, and therefore able to handle a further increase of the global routing table. Furthermore, measurements on real and recent devices show that the introduced delays for flows passing the software switch are acceptable.

Contents

List of Figures

List of Tables

Listings

1 Introduction

1.1 Motivations and Contributions

In the last decades the internet has grown dramatically in every aspect. The traffic amount, the number of users and the number of websites, just to name a few, have all multiplied. This increase causes much higher requirements for today's IP routers.

Software Defined Networking (SDN) is a new approach for networking. It separates control and data plane to improve the controllability of a network by using a SDN controller to fully control each network device. This brings new opportunities for researchers and network operators.

In [3] M. A. Chang et al. proposed a technique to boost, or *supercharge*, the convergence time of IP routers by combining them with a SDN switch. This project is a closely related project as it aims to improve router performance with SDN. However, while [3] focuses on convergence time, we focus on another aspect that suffers from the internet explosion: the Forwarding Information Base (FIB) size.

This project provides a way to handle the increasing number of prefixes which have to be stored in an IP router's FIB without upgrading or replacing a router, just by adding a SDN switch and a Controller. The proposed algorithm splits the FIB and stores only parts of it directly in the router, while the other prefixes are stored in an SDN enabled software switch. Furthermore, the provided algorithm uses real-time traffic statistics to optimize the allocation of the routes. To do so, the prefixes are sorted according to their usage and split, so that the most used routes are stored in the router's FIB while the remaining routes are in the switch's forwarding table. Additionally, the dependencies of nested prefixes are taken into account when calculating the route split to prevent incorrect forwarding.

1.2 Related Work

The rapid growth of the internet routing table and the resulting problems of increased requirement in FIB size is well known and subject of many papers. In our project we use the common approach of route caching, whose benefits K. Gadkari et al. show in [4], but there are other approaches like route aggregation. Y. Liu et al. show that route aggregation can reduce the FIB size in [5] and introduced different aggregation algorithms in [6] and [5].

There are multiple papers about caching of routes. D. Feldmeier shows in [7] shows, that caching can be used to reduce the lookup time per packet and therefore increase the throughput. H. Liu provides a network processor design for caching route prefixes and shows that prefix caching is more effective than caching of single IP addresses in [8]. In [9] C. Kim et al. study caching of flat, uni-class prefixes by splitting the advertised prefixes in /24 subnets. Our solution has similarities in using a cache-like structure, but differs in deploying different devices and protocols for cache (router) and main memory (software SDN switch). Moreover, with our solution it's not necessary to change the router hardware.

When caching route prefixes, it's necessary to prevent cache-hiding (see section 2.1). [9] handles this problematic by using only /24 prefixes. Another approach is proposed by Y. Liu et al. in [10]. They solves this problem as they calculate the minimal non-overlapping part of the prefix to store in the cache. In our solution we use a different approach where we only cache routes along with all covered prefixes with different next hops.

On the SDN side, N. Katta et al. provides a software-hardware hybrid, where OpenFlow rules are allocated either to a hardware- or software-switch in [11]. We use the same idea of combining software and hardware devices. The difference is that we use different device types, a software SDN switch and a physical IP router, while [11] uses hardware and software SDN switches. Another example FIB splitting in SDN is [12] by H. Ballani et al: they proposed to split the FIB among different forwarding devices in a SDN network. Compared to our solution [12] has the drawback that they need an already deployed SDN network, while our solution also works with previously non-SDN networks.

1.3 Background

1.3.1 Software Defined Networking and OpenFlow

Software Defined Networking (SDN) [13] is a promising approach to improve today's communication networks. It separates the control- and data-plane and provides forwarding, state distribution and specification abstractions.

Today's Off-the-Shelf networking devices are closed-box devices consisting of specialized packet forwarding hardware, an operating system and different features on top of it. In SDN, the operating systems are centralized in a SDN Controller, which controls the whole network. Moreover, the controller provides a well-defined API, which can be used by networking applications.

OpenFlow [14] is a protocol, used by SDN controllers to manage OpenFlow capable switches. It allows to remotely add or remove entries to the switch s forwarding table over a secure TCP connection. There are other protocols for this purposes, but OpenFlow is mostly used and therefore we decided to use it for our project.

1.3.2 Internet, AS and BGP

The Internet, as we know it today, is a network of networks. Those networks are so called *autonomous systems* (AS) [15]. The Border Gateway Protocol (BGP) [16] is a protocol used for route and reachability information exchange between those ASes. Between two manually configured BGP neighbors a TCP Session on Port 179 is established to exchange routing information. Each participant stores this routing information and advertises them to all its neighbors. A BGP route contains route information of which we use the router prefix, next hop and AS path.

The routers store those advertised routes in their RIB. The Forwarding Information Base (FIB), used to forward packets, is directly derived from the routes stored in the FIB. If a new packet arrives at a router, the router will forward it to the next hop of the route with the longest matching prefix. Today's global routing table consists of more than 500k prefixes. To handle all this routes, a huge FIB size is required.

1.3.3 NetFlow

NetFlow [17] is a common network device feature which is used to monitor IP traffic. The device (flow exporter) collects flow records and exports this real-time data towards a client application (flow collector). This exported data can be used to analyze network traffic. In our project the router is the flow exporter and we use this functionality to export the number of bytes transmitted to each destination IP to optimize the FIB split between router and SDN switch.

1.4 Terminology

As mentioned before, the advertised routes are stored in the routers RIB. The FIB entries, which are used to actually forward the packets, are directly related to the RIB entries. Since the FIB has to be implemented in fast, and therefore expensive, memory this is normally the limiting part for the numbers of routes a router can handle. For simplicity we only mention the FIB throughout this report even though both, the RIB and FIB sizes of the router, are increased by our solution.

2 Design

2.1 Overview

As mentioned before, this project aims to reduce an IP routers FIB size. We want to accomplish this without buying new routers or upgrade existing ones. Furthermore, the provided solution should work with any conventional BGP neighbor.

To accomplished this, we add a layer-2 switch connected to a server in between the router and its neighbors. The server is running a software SDN switch and SDN controller, a BGP daemon and a NetFlow collector. Instead of the direct connection between our IP router and its BGP neighbors we run the BGP daemon on the server to take control over the BGP messages advertised to our router and pass them to our *FIB split algorithm* (see section 2.5). We announce as many prefixes to our router as it has space in its FIB[1] minus one. This last entry we use to add a default route which forwards all traffic towards prefixes not present in the router's FIB to the software switch. All remaining routes are stored in the switch's forwarding table.

This setup could generate a huge load on the software switch if a lot of traffic has to pass it. To prevent this we introduce an optimization function, which periodically optimizes the route allocation by using the available real-time NetFlow data. This means we sort all route prefixes according to their usage and split them so that after the optimization the most used routes are stored in the router's FIB.

Another problem we have to encounter is *cache hiding*. This means, that packets could be wrongly forwarded because a route stored in the router's FIB is hiding the correct route stored in the forwarding table of the switch (see figure 1). Since routes are executed in a longest matching prefix fashion, this can happen if nested prefixes have different next hops and are stored in different places. To handle this we always calculate the route dependencies of nested prefixes when we announce new routes or optimize the route allocation. For example in the situation showed in figure 1 we make sure that the prefix 1.0.0.0/24 is stored in the router as well so that a proper forwarding is assured.

In the following, it's described in detail how we *supercharge* the router FIB size. Even though it's designed to work with any reasonable amount of BGP neighbors, we concentrate on the setup shown in figure 2. R1 is the router we *supercharge*, whereas R2 and R3 are its BGP neighbors, which advertise a total of more than 500k prefixes to R1.

2.2 Proposed Network Architectures

As one of the first things, we had to decide which network architecture we use to increase the router's FIB size. This consists of what type of devices we use and how to arrange them. We use the term *controlling software* in this section to summarize all controlling elements, namely the SDN controller, BGP daemon, NetFlow collector and the provide

[1] The FIB size of the router has to be set in the configuration file of the algorithm.

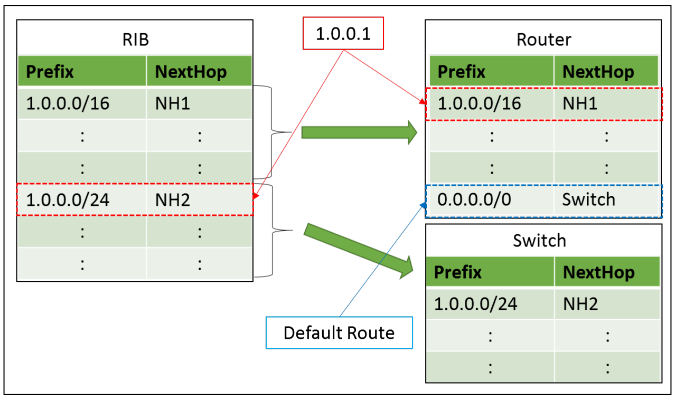


Figure 1: Example of erroneous splitting, which results in different longest matching prefixes of the IP 1.0.0.1 before and after the splitting of the FIB and therefore in incorrect forwarding.

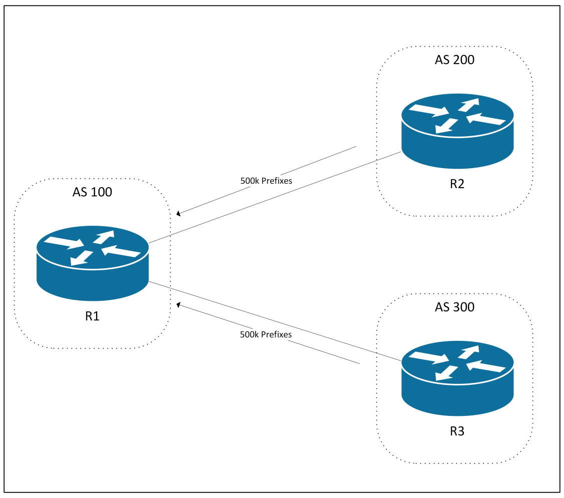


Figure 2: Initial Setup: R1 is the router we *supercharge*, while the adjacent routers R2 and R3 represent its BGP neighbors.

algorithm, which we explain in section 2.5. We only focus on the network devices and their connections in this section. The proposed network architectures are:

1. **Single software SDN switch with additional links**
   The first idea was to add a single, software SDN switch connected to all three routers, while keeping the original links (see figure 3). The advantage of this architecture is that there is no delay for all packets sent to a prefix stored in the router's FIB. The one, but rather big, disadvantage is the two additional links needed between the neighboring routers and the software switch. Since this work shouldn't change anything related to R2 and R3 the idea was discarded.

2. **Single physical SDN switch**
   The next proposed architecture was a single, physical SDN switch between all three routers (see figure 3). Because of the limited number of flow entries in a physical SDN switch, this setup would be similar to a 2-layer cache, consisting of the router, switch and controller. The resulting computation complexity to prevent cache hiding would be significantly higher, so this architecture was discarded as well.

3. **Software SDN switch and physical layer-2 switch**
   The last, and eventually chosen, proposed architecture consists of a physical layer-2 switch combined with a software SDN switch (see figure 4). All traffic towards prefixes stored directly in the router would have no delay, while the rest of the traffic would pass the software switch as an additional next-hop. Regarding [2], the additional next-hop should only affect an insignificant amount of the traffic passing the network if we periodically optimize the distribution of the routes between the router and the software switch.

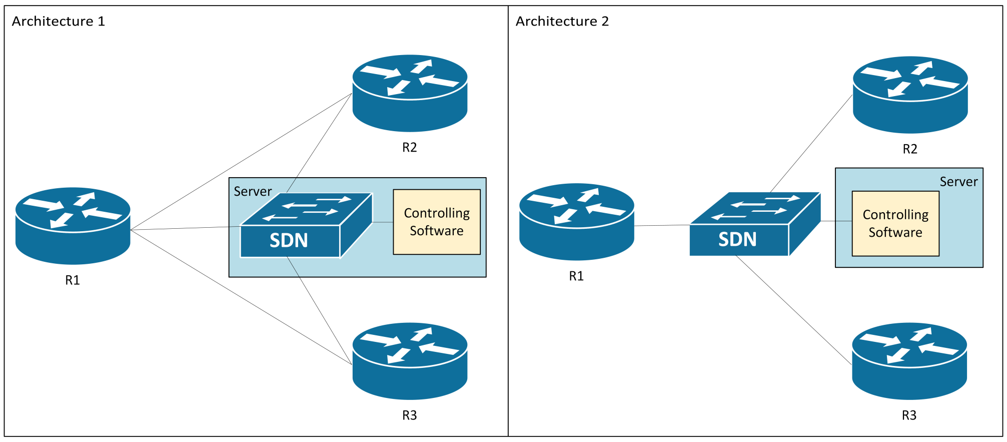


Figure 3: Discarded network architectures

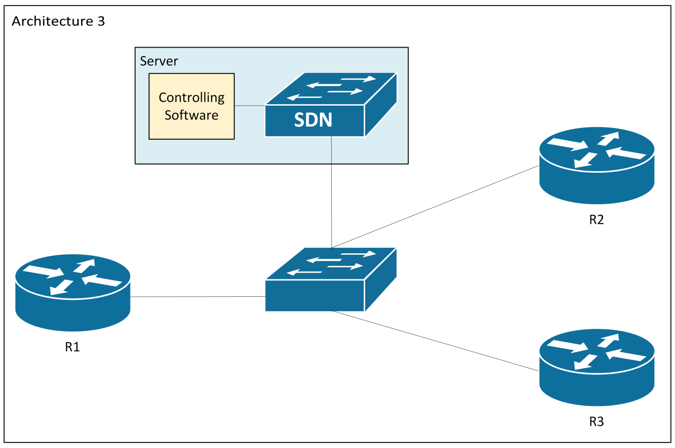


Figure 4: Chosen network architecture

2.3 Setup

The final setup is shown in figure 5. The three routers and the server, running the software switch and the controlling software, are connected by the layer-2 switch. A detailed description of each hardware and software component and its configuration follows in the next sections.

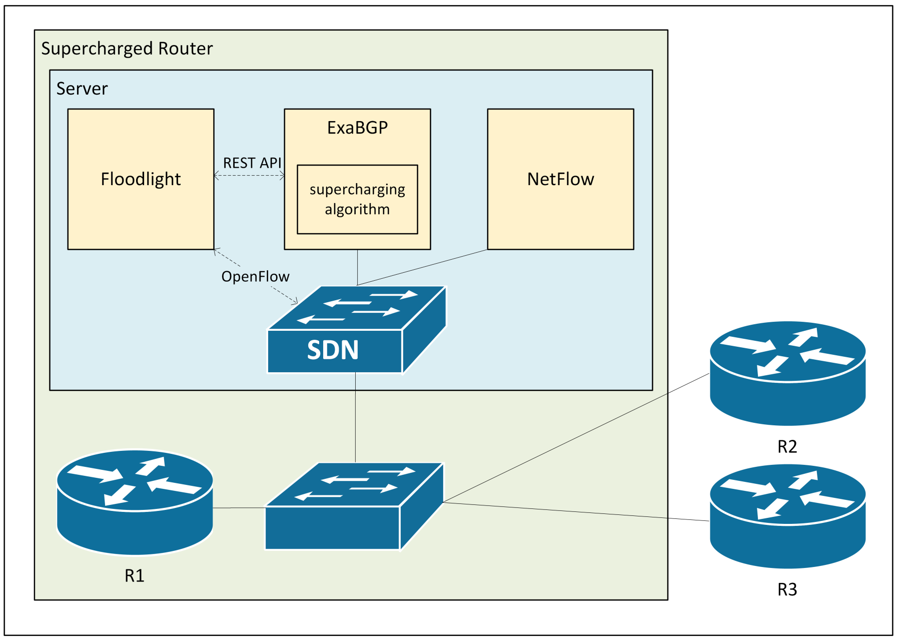


Figure 5: Supercharged Router

2.3.1 Router and Switch

Even though our solution is designed to work with any BGP capable router, which additionally offers NetFlow data export, we had to choose which one we use for our tests. We chose to use recent and widely used Cisco routers. To prepare the router we configure it with an arbitrary AS number and our BGP daemon with the private AS number 65000 as BGP neighbor. Furthermore we enable NetFlow on the interfaces and configure it to export the traffic statistics to our NetFlow collector on port 10123[2].

The neighboring routers (R2 and R3 in Figure 5) only need to be configured as BGP neighbors of the BGP daemon.

Since the only purpose of the hardware switch is layer-2 forwarding, we don't use any special configuration.

2.3.2 SDN Switch and Controller

We decided to deploy an Open vSwitch (OVS) [18] combined with a Floodlight controller [19], since they satisfy our requirements and are open-source.

OVS is an open source software switch. Its main function is to forward traffic between physical and virtual network interfaces on a host. Furthermore it provides different methods for network control systems, of which we used OpenFlow, to insert rules into the switches forwarding table. The OVS is configured to connect to the Floodlight controller on port 6633. Additionally we virtually added the servers network interface to the switch, so all incoming traffic will pass it. At last we set the fail-mode to *secure* to make sure, that the SDN controller always has full control over the forwarding behavior of the OVS.

Floodlight is an open source SDN Controller. It provides an easy-to-use REST API which we use to add entries to the Forwarding Table of the OVS. Naturally, the controller is configured to listen on the port we configured in the OVS. Furthermore, we use the default modules to provide the functionality for basic forwarding and adding forwarding rules via its REST API [20].

The REST API is accessed via appropriate HTTP GET, HTTP POST and HTTP DELETE requests to the controller. To add or delete OpenFlow rules to the OVS, the Static Flow Pusher module [21] of Floodlight is accessed via the REST API. To insert an OpenFlow rule the rule has to be added in JSON format to the data part of a HTTP POST. To remove a rule a HTTP DELETE has to be sent.

2.3.3 ExaBGP

As BGP daemon we use the networking tool ExaBGP [22], which transforms BGP messages into JSON or text format and let you use your own scripts to process this messages as well as sending out your own BGP messages via its API.

A configuration file is used to configure ExaBGP. In this file we add the three routers as BGP neighbors and activate the forwarding of the BGP messages, like update or state messages, to our algorithm.

To access the received BGP messages and generate the outgoing messages, we use ExaBGP's API. We can read the incoming messages in JSON format by reading from standard input in our algorithm. To send messages we have to write to standard output. We use two types of BGP messages in our algorithm: announce messages and withdraw messages (see listing 1 for an example).

[2] The port 10123 is freely chosen and can be changed, as long as it's done for all related components.

Listing 1: Example of ExaBGP's API

```
# BGP announce message
stdout.write('neighbor 10.0.0.1 announce route 24.28.189.0/24 next-hop
    10.0.0.2 as-path [200 3957 999]')

# BGP withdraw message
stdout.write('neighbor 10.0.0.1 withdraw route 24.28.189.0/24 next-hop
    10.0.0.2')
```

2.3.4 Nfdump

We use the NetFlow functionality of the router to export traffic statistic form the router. To receive this data and write it to a file in a readable form for our algorithm we use nfdump [23]. Nfdump is a command line tool to process NetFlow data. It consists of different functions. We use nfcapd to capture the traffic statistics exported by the router and nfdump to write it to suitable format which can be read by our algorithm.

We configure nfcapd to listen on port 10123 and call nfdump to write the IP destination and the number of Bytes transmitted to a file each 60 seconds. An example of one line in this file is: *1.0.25.94,9583*. The part left of the comma is the IP destination, whereas the part on the right is the number of bytes transmitted to this destination. In a later step, the algorithm assigned the destination IPs to the longest matching prefix and sums up the number of bytes transmitted for each prefix.

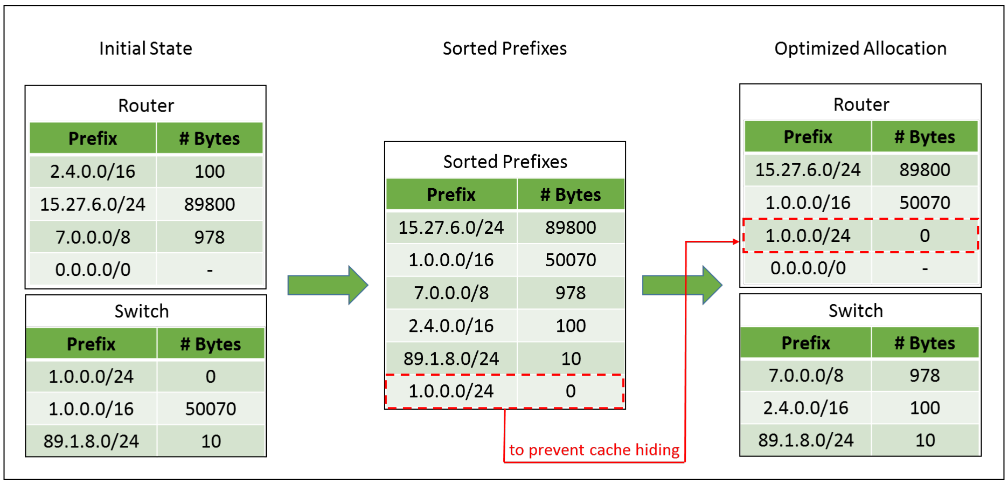
2.4 Supercharging an IP Router's FIB size

When we start up the *supercharged router*, ExaBGP starts the *FIB split algorithm* and establish a TCP connection for BGP with each routers. The BGP messages are now forwarded via ExaBGPs API to our algorithm in JSON format. The algorithm examines the messages and handles them according to their content. Depending on this information appropriate actions are taken, for example sending update messages to announce routes to the routers or just store the information. To increase the router's FIB size, we only announce as many routes to the router as it has space in its FIB minus one. This one space left, is used to add a *default route* 0.0.0.0/0 with the OVS as next hop. Since a router forwards packets in a longest matching prefix fashion, only packets sent towards prefixes not stored in the FIB are forwarded to the OVS, while the others are forwarded directly to the correct next hop. As soon as the router's FIB is full, incoming routes are added to the OVS' forwarding table. While communicating with all routers is done via ExaBGP's API, storing routes as OpenFlow rules in the OVS' forwarding table is done via Floodlight's REST API.

Right after the BGP connections are established, the BGP neighbors start to announce all known routes to the *supercharged router*. This generates a huge number of BGP messages. Since the communication buffer between ExaBGP and our algorithm is limited, the time to handle each message is limited as well. To prevent buffer overflows we implemented a *burst mode* which starts if more than a certain threshold of new routes are announced in a few seconds. Once the algorithm enters the *burst mode* it only announces new routes to the neighbors but not to the supercharged router or OVS.

Instead, the routes are temporarily saved for later announcements. The reason for this is that the computation of the dependencies and adding of rules to the OVS don't satisfy the timing constraints. As soon as the input buffer is empty and no new messages arrive for a few seconds the *clean-up* of the *burst mode* starts. This means all temporarily stored routes are now processed and finally added either to the router or the OVS.

After the start-up phase the route allocation between the router and the OVS is periodically optimized to minimize the traffic passing through the OVS. To do so, the algorithm periodically reads the NetFlow data exported by the router and sorts the route prefixes according to their usage. After that, the algorithm starts swapping rules in a way that after the optimization the most heavily used prefixes are stored in the supercharged routers FIB, while the others are installed as flow rules in the OVS' forwarding table (see figure 6).



Router (Initial State)

Prefix	# Bytes
2.4.0.0/16	100
15.27.6.0/24	89800
7.0.0.0/8	978
0.0.0.0/0	-

Switch (Initial State)

Prefix	# Bytes
1.0.0.0/24	0
1.0.0.0/16	50070
89.1.8.0/24	10

Sorted Prefixes

Prefix	# Bytes
15.27.6.0/24	89800
1.0.0.0/16	50070
7.0.0.0/8	978
2.4.0.0/16	100
89.1.8.0/24	10
1.0.0.0/24	0

Router (Optimized Allocation)

Prefix	# Bytes
15.27.6.0/24	89800
1.0.0.0/16	50070
1.0.0.0/24	0
0.0.0.0/0	-

Switch (Optimized Allocation)

Prefix	# Bytes
7.0.0.0/8	978
2.4.0.0/16	100
89.1.8.0/24	10

Figure 6: Example of an optimization of the route allocation with prevented cache hiding.

To prevent cache hiding, the *FIB split algorithm* always checks route dependencies between all known prefixes and only announces routes to the router if all dependent routes can be added to, or are already in, the router's FIB.

2.5 FIB Split Algorithm

The *supercharging algorithm* is written in python and contains all functions needed to split the FIB. We only show the most important functions here. For the whole code see the Github repository[3].

2.5.1 BGP Handling

The core of the *FIB split algorithm* is the BGP handling. As the name says it handles incoming BGP messages and generates all outgoing messages by calling the appropriate functions. Figure 7 provides an overview of the message processing.

[3] `https://github.com/nsg-ethz/supercharged_memory`

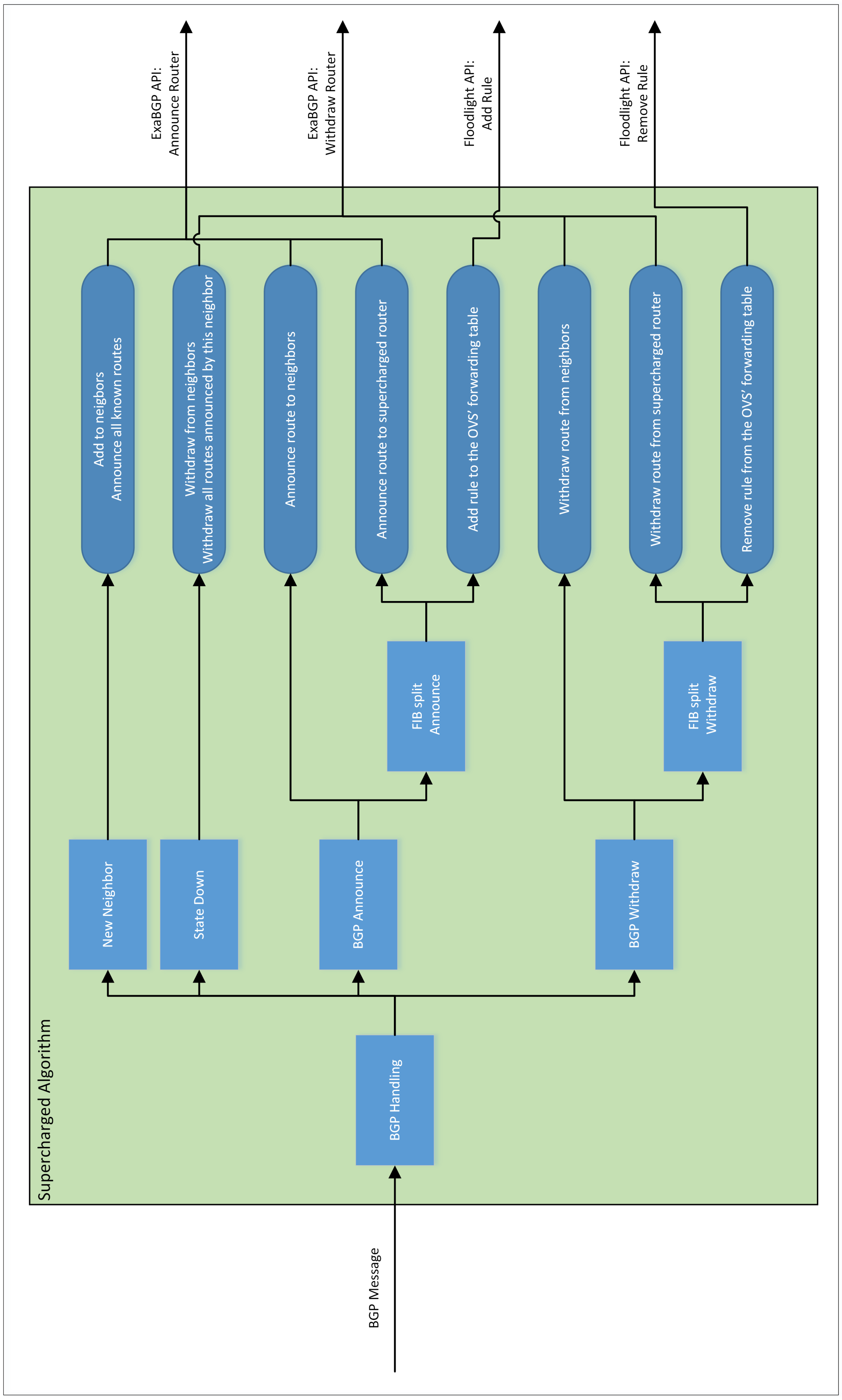

BGP Message
Supercharged Algorithm
BGP Handling
New Neighbor
State Down
BGP Announce
BGP Withdraw
FIB split Announce
FIB split Withdraw
Add to neigbors Announce all known routes
Withdraw from neighbors Withdraw all routes announced by this neighbor
Announce route to neighbors
Announce route to supercharged router
Add rule to the OVS' forwarding table
Withdraw route from neighbors
Withdraw route from supercharged router
Remove rule from the OVS' forwarding table
ExaBGP API: Announce Router
ExaBGP API: Withdraw Router
Floodlight API: Add Rule
Floodlight API: Remove Rule

Figure 7: Overview of the BGP message handling and the resulting calls of the two APIs.

The main structure used by the algorithm to store all known routes is the python dictionary *fib*. The keys are the known route prefix, while as values, we use a list of known *bgp_routes* to this prefix. The structure *bgp_routes* is defined by a route prefix, next hop, AS-path and AS-path length and also provides a *compare* function which considers a route better than another if the AS-path is shorter. If the AS-path length is the same the decision is based on the IP address of the next hop. The first element of the list of *bgp_router* is always the best known route to this prefix. Similar to the *fib*, *fib_switch* and *fib_router* are used to keep track of the route split.

The BGP handling uses ExaBGP's API to receive and send BGP messages. This basically means reading from standard input to receive new messages and write to standard output in a specific form to send messages. If there is a new message the algorithm calls different functions to handle the parts of the message we are interested in (see Listing 2).

The first part of a BGP message always contains the senders IP. If this IP has never been seen before, it is added to our neighbor list and all known BGP routes are announce to the new neighbor[4]. Upon failure of a router, a *state down* notification is sent from ExaBGP to the algorithm. If such a message is received, the algorithm withdraws all routes advertised by this router and removes the router from his neighbors.

Listing 2: Main Loop

```
# All known rules are stored in the fib dictionary. The keys are
# the prefixes, while the values are lists of all known routes to
# this prefix. The first element is always the best known route.
fib = {}

# time of the last optimization
timeOptimization = time.clock()

# Burst mode variables:
# BurstMode indicates if we are currently in burs mode and
# timeBurst is the time of the last check.
burstMode = False
timeBurst = time.clock()

while True:
    # read incoming BGP messages and handle different types
    msg = read.bgp_message()

    # If msg is empty, no new messages are in the input buffer
    if not msg:
        if burstMode:
            endBrustMode()
            optimizeRouteAllocation()

        elif timeOptimization > optimizationInterval:
            optimizeRouteAllocation()

        timeOptimization  = time.clock()
        continue

    # Enter burst mode if necessary
    if not burstMode and time.clock() - timeBurst > timeThreshold:
```

[4] In the following, by neighbor we refer to any adjacent router, excluding our *supercharged* router

```
        if len(fib) - numRoutesOld > burstThreshold:
            burstMode = True
        numRoutesOld = len(fib)
        timeBurst = time.clock()

    # If the senders IP has never been seen before add it as
    # neighbor and announce all known routes to it
    if not (msg.senderIP in neighbors):
        neighbors.append(msg.senderIP)
        announceAllKnownRoutesToNeighbor(msg.senderIP)

    # If it is a state down message withdraw all routes
    # announced by this neighbor
    if msg.senderState == 'down':
        neighbors.remove(msg.senderIP)
        for bgpRoute in fib:
            if bgpRoute.nextHop == senderIP:
                bgpWithdraw(bgpRoute)

    # Examine the "update" part of the message if there is one
    if msg.update and  'announce' in msg.update:
        bgpAnnounce(routes)
    elif msg.update and 'withdraw' in msg.update:
        bgpWithdraw(routes)
```

Next, the route announce and withdraw is handled. The two functions BGP announce (listing 3) and BGP withdraw (listing 4), respectively, are called to handle this part and distribute the new information among all routers. For the case, that more routes than a certain threshold are announced in few seconds, a "burst mode" is implemented to prevent a buffer overflow on the input buffer.

Listing 3: BGP Announce

```
def bgpAnnounce(routes):
    for route in routes:
        # If there is already a route for the same prefix only announce if
        # the new route is better than the previously best known route.
        # The best known route is always the first element of the list
        # containing all routes towards this prefix (fib[prefix][0])
        prefix = route.routePrefix
        if prefix in fib and route.isBetter(fib[prefix][0]):
            fib[prefix].append(route)
            continue

        fib[prefix].insert(0,route)
        for neighbor in neighbors:
            if neighbor == supercharged.IP:
                if not burstMode:
                    # Call separate function for the fib splitting
                    fibSplitAnnounce(route)
                else:
                    # In burst mode only store it to save time
                    tempRoutes.append(route)
            else:
                # Use ExaBGPs API to announce the route to all neighbors
                stdout.write(<announce route to neighbor>)
    return
```

Listing 4: BGP Withdraw

```
def bgpWithdraw(routes):
    for route in routes:
        prefix = route.routePrefix
        fib[prefix].remove(route)

        # If there is still another route for this prefix, announce
        # the new best route, else withdraw the route
        if fib[prefix]:
            bgpAnnounce(bestKnownRoute)

        else:
            for neighbor in neighbors:
                if neighbor == supercharged.IP:
                    # Call separate function to handle our router
                    fibSplitWithdraw(route)
                else:
                    # use ExaBGPs API to withdraw the route
                    stdout.write(<withdraw route from neighbor>)

    return
```

If there aren't any new BGP messages in the input buffer the clean-up function of the *burst mode* is called, which announces all previously stored routes to our router by calling the *FIB split announce* function for each of them(see section 2.5.2). Furthermore, the optimization function of the *FIB split algorithm* (section 2.5.3) is called periodically to reallocate the stored routes according to their usage.

2.5.2 FIB Split Announce and Withdraw

To announce routes to our router, the *FIB split announce* function is called (see listing 5). This function checks if there is still enough space in the router's FIB to store the new route or if it's necessary to use the OVS's forwarding table. Additionally, it checks if any of the previously announced prefixes cover, or is covered, by the new one and swaps prefixes between router and OVS if it's necessary to prevent cache hiding.

Listing 5: FIB Split Announce

```
def fibSplitAnnounce(newRoute):
    # If the prefix is already known, the route is replaced
    if newRoute.routePrefix in fib:
        fib[newRoute.routePrefix] = newRoute
        if newRoute.routePrefix in fib_router:
            fib_router[newRoute.routePrefix] = newRoute
            announceToRouter(newRoute)
        else:
            fib_switch[newRoute.routePrefix] = newRoute
            announceToSwitch(newRoute)

    else:
        fib[newRoute.routePrefix] = newRoute

        # As long as the fib switch is empty and there is still router
        # memory left, there is no need for dependency checks
        if len(fib_switch) == 0 and len(fib_router) < routerMem - 1:
                announceToRouter(newRoute)
                fib_router[newRoute.routePrefix] = newRoute
                return

        else:
            # check if the new route interferes with known routes
            if checkDependancies(newRoute.routePrefix):
                # rearrangeRoutes swaps routes between router and switch
                # and returns true or false to indicate whether the route
                # must be stored in the OVS
                switch = rearrangeRoutes()

            if not switch and len(fib_router) < routerMem -1:
                announceToRouter(newRoute)
                fib_router[newRoute.routePrefix] = newRoute
            else:
                if not defaultRoute:
                    announceDefaultRouteToRouter()
                announceToSwitch(newRoute)
                fib_switch[newRoute.routePrefix] = newRoute
            return
```

The *FIB split withdraw* function checks whether the route is stored in the router's FIB or the OVS' forwarding table and uses the appropriate API to withdraw the route (see listing 6).

Listing 6: FIB Split Withdraw

```
def fibSplitWithdraw(route):
    # Withdraw route from where it's stored
    del fib[route.routePrefix]
    if route.routePrefix in fib_switch:
        withdrawFromSwitch(fib_switch[route.routePrefix])
        del fib_switch[route.routePrefix]
    else:
        withdrawFromRouter(fib_router[route.routePrefix])
        del fib_router[route.routePrefix]
    return
```

2.5.3 Optimization

The optimization function is used to maximize the *supercharged router's* throughput. It aims to fill the router's FIB with the most used routes and store the less used in the OVS' forwarding table. The function consists of two parts: the reading of the NetFlow data (see listing 7) and the swapping of the prefixes (see listing 8)

First, the NetFlow data is read. It consists of *IP, # Bytes* pairs for each recorded flow. For each flow, the number of bytes transmitted is now added to the longest matching Prefix. After that, an ordered list consisting of all known prefix is generated and ordered according to the number of bytes per prefix. According to this list, the function now splits the prefixes so that we get a list with the most used prefixes to fill the router's FIB and another one with all remaining prefixes for the OVS' forwarding table. While generating these two lists, the algorithm constantly calculates the prefix dependencies and takes them into account: a route is only put into the prefix list for the router if all covered routes with different next hops are either already in this list or there is enough space left to put all of them in this list.

Listing 7: Read NetFlow Data

```
def supercharge.readNetFlowData():
    # Read data and sort prefixes in an ordered List
    data = {}
    for ip,bytes in readlines(newFlowData.txt).split(","):
        # search best returns the best fitting known prefix
        prefix = searchBest(ip)
        data[prefix] += bytes

    prefixList = sortPrefixesByBytes(data)

    # Add known prefixes, first from fib_router, then fib_switch to
    # prevent unnecessary changes
    for prefix in fib_router:
        if not prefix in prefixList:
            prefixList.append(prefix)

    for prefix in fib_switch:
        if not prefix in prefixList:
            prefixList.append(prefix)

    # Split the prefixList to different fibs,
    # while taking care of dependencies
    prefixRouter = []
    prefixSwitch = []
    for prefix in prefixList:
        # If the prefix is already allocated -> continue
        if prefix in prefixRouter or prefix in prefixSwitch:
            continue

        # If no, search all dependent prefixes with different next Hops
        # and add them to one of the lists
        depGroup = getDependancyGroup(prefix)
        if  len(depGroup) + len(prefixRouter) < routerMem -1:
            prefixRouter.append(depGroup)

        else:
            prefixSwitch.append(prefix)

    return prefixRouter,prefixSwitch
```

Once the two prefix lists for the router and the OVS are generated the swapping of the prefixes takes place. First, the prefixes which are in the list for the OVS are added to the OVS' forwarding table if they aren't already there. After that, the prefixes not in the router's list are removed from the router's FIB. Then there is exactly enough space left in the router's FIB to add the missing prefixes from the router's list. And in the end, the redundant routes in the OVS' forwarding table are removed. By swapping the prefixes in this manner there is a route to each prefix either in the OVS or the router at any time.

Listing 8: Optimize Route Allocation

```python
def optimize:
    # Optimize route allocation according to the available traffic data.
    newSwitchFib,newRouterFib = readNetFlowData()

    # First add prefixes to the OVS.
    for prefix in newSwitchFib:
        if not prefix in fib_switch:
            announceToSwitch(fib[prefix])
            fib_switch[prefix] = fib[prefix]

    # Then remove routes from the router's FIB.
    for prefix in fib_router:
        if not prefix in newRouterFib:
            withdrawFromRouter(fib[prefix])
            del fib_router[prefix]

    # After that, there is space for the new prefixes in the router.
    for prefix in newRouterFib:
        if not prefix in fib_router:
            announceToRouter(fib[prefix])
            fib_router[prefix] = fib[prefix]

    # At last, remove the redundant routes from the switch.
    for prefix in fib_switch:
        if not prefix in newSwitchFib:
            withdrawFromSwitch(fib[prefix])
            del fib_switch[prefix]
```

3 Evaluation

We use two different techniques for the evaluation: an offline simulation and a prototype setup with real and recent devices in the lab. We used the simulation to evaluate the performance of the *FIB split algorithm* by measuring execution times of the *optimization* (see section 2.5.3) and the time it takes the *FIB split announce* (see section 2.5.2) to announce new routes. On the lab setup, we measured the packet delays by replaying real traffic traces and the time it takes to announce new routes.

It has to be said, that while the simulation of the algorithm worked without names worthy complications, we encountered some problems with the prototype setup. That caused everything to take much more time than expected and therefore we did not have enough time to evaluate the optimization function in the lab. These problems are explained in section 3.2.2.

3.1 Simulation

To measure execution times for different functions we used an offline simulation. That means, we don't actually run any of the components, instead we used a simulation script which calls different functions of our algorithm and measured the execution times. The observed functions were the *FIB split announcement* and the *optimization*. We used BGP route information from [24] for all executed simulation runs.

The simulation script does the following:

1. Read BGP routes from file
2. Runs for different numbers of routes and FIB size:
   a) Route announcements
   b) Multiple times the optimization with randomly generated NetFlow data

Here, we only focused on the execution times of the functions not the time needed to actually announce the routes to the router or install it as an OpenFlow rule in the OVS's forwarding table.

3.1.1 Route Announcement

We ran the simulation for this measurement with a fixed number of 558k routes and different FIB sizes and recorded the execution time for each announcement. We assumed, that the values higher than 100 times the average are spikes caused by background tasks of the laptop OS and removed them (approximately 1 per 50k).

Figure 8 shows the total time needed to announce the number of routes in the x-axis. Here, we see that the total time to announce the routes is approximately linear to the number of routes.

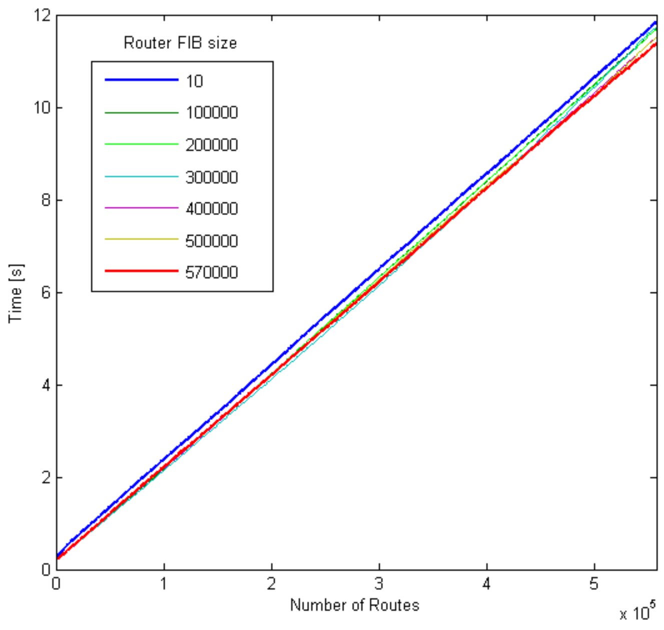


Figure 8: Summarized execution time for route announcements.

In figure 9 we show the time it takes to add one new route. Each blue dot represents the average of 1000 new routes. For example a dot with *x,y* values *22000,0.023* means that it takes an average of 0.023 ms to add a new route when already 21001 to 22000 routes are in the *supercharged router*. The red line shows the average to add a new route before and after the FIB is filled.

Regarding this results, the following statements can be made:

1. Adding new routes takes slightly more time as soon as the FIB is full and
2. the announcement time is independent of the total number of routes already stored.

The first one is easily explained and was what we expected: As long as all known routes can be stored in the router's FIB the function doesn't have to calculate any dependencies and is therefore faster. This check is implemented right at the beginning of the function, so as long as the OVS doesn't hold any rules and there is still space in the FIB no further calculations have to be made.

The second statement follows from the linearity observed in the first graph. This is mandatory to handle the increasing amount of BGP routes in the Internet.

3.1.2 Optimization Function

For this measurement we configured the simulation to announce different amounts of routes and FIB sizes. As soon as all routes are announced random NetFlow data is generated and the optimization function is called a few times.

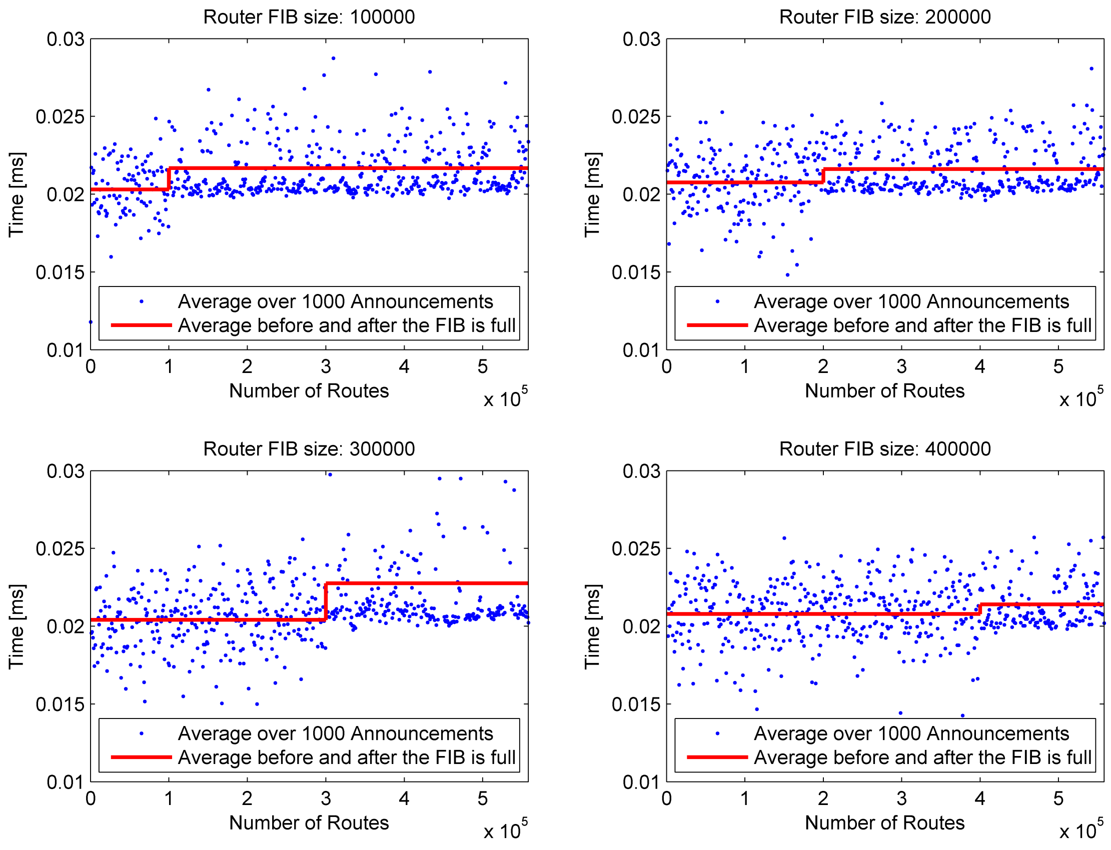


Figure 9: The four graphs show the time needed to announce a new route to the supercharged router.

The data for the presented measurement consists of ten optimization calls. The line shows the average, while the markers show the recorded maximum and minimum execution times. We removed the measurements for FIB sizes bigger than the number of routes since in that case the optimization functions does not have to calculate anything and therefore finishes in a few milliseconds.

The recorded execution time is depicted in Figure 10. It can be seen that the execution time is approximately linear to the number of routes for a given FIB size. Similar to the linearity of the recorded announcement time in the previous section, this is a mandatory requirement for the *supercharging algorithm* to withstand today's huge amount of BGP routes. The increasing execution time for a fixed number of routes and increasing FIB size is due to the fact, that the algorithm first fills the FIB and has to consider all dependent routes. After the FIB is filled, all remaining prefixes can safely be added to the OVS without caring about dependencies.

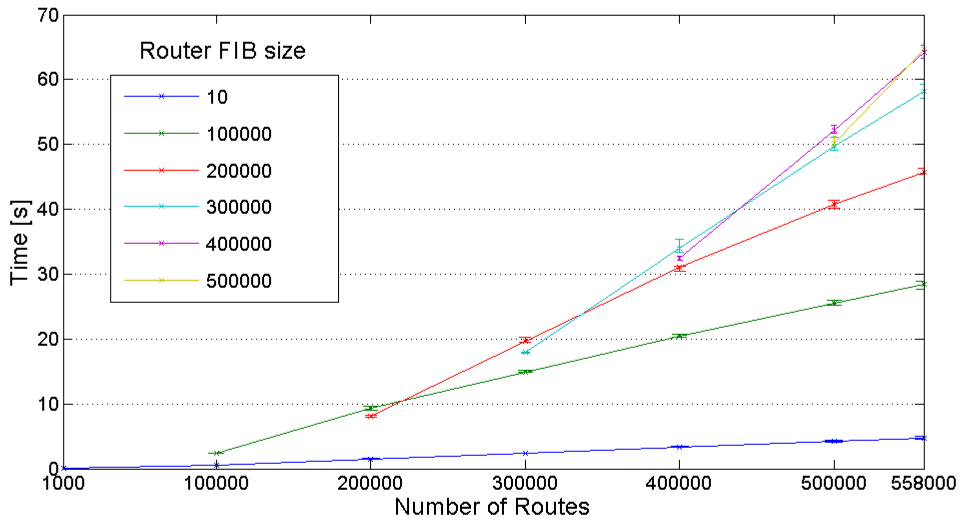


Figure 10: The graph shows the average execution time of the optimization function for different FIB sizes and number of routes.

3.2 Measurement

3.2.1 Lab Setup

The complete set up we use for the evaluation is depicted in figure 11. It consists of a HP E3800 J9575A, which we use as layer-2 switch, and two Cisco Nexus 7k C7018 (NX-OS v6.2) running our three routers as virtual devices. The three layer-2 switches in the scheme are on the same physical switch, separated by vlans. To run the necessary software for the supercharged router, as well as the software used for the measurement, we used three laptops (ubuntu 14.04). *Laptop3* in figure 11 builds the core of the *supercharged router*. It runs the OVS, SDN controller, ExaBGP, nfdump and our algorithm. The other two Laptops are used for the measurements. We use *Laptop1* to send traffic through the network and *Laptop2* to announce BGP routes to R2 and R3.

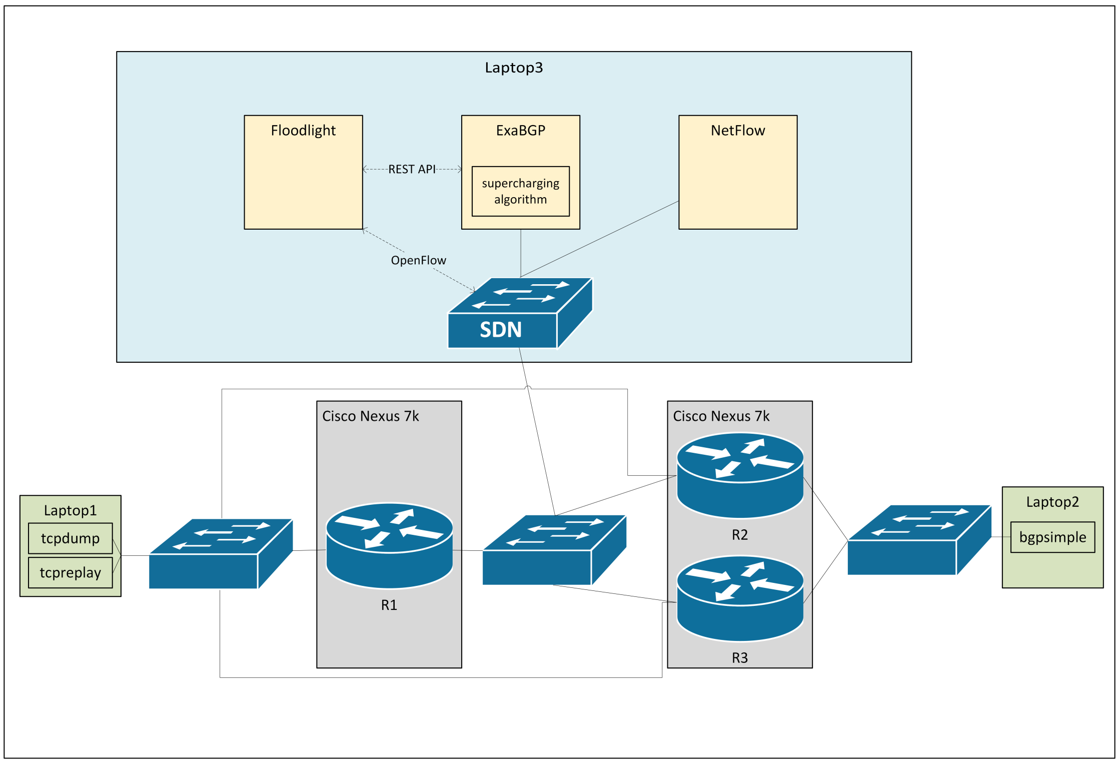


Figure 11: Lab Setup

In addition to the software used for the *supercharged router* described in section 2.3, we used bgpsimple to advertise BGP routes to R2 and R3. To send traffic and record the incoming packets we use tcpreplay and tcpdump.

Bgpsimple Bgpsimple [25] is a perl script used to announce BGP routes to adjacent routers. The input is a file in a specific format with all the needed BGP route information. After starting the script it first establish s a connection with the configured neighbor and then advertises all routes contained in the file.

Tcpdump Tcpdump [26] is a networking tool which can be used to record all traffic passing the specified networking interface. The recorded data can be exported to a *.pcap* file.

Tcpreplay Tcpreplay [27] is a software suite, which provides multiple tools to test different network devices. For our measurements we use tcprewrite and tcpreplay. We use tcprewrite to prepare the packet headers of our traffic traces for the measurement, namely change the Ethernet destination MAC address to the one of router R1. After the preparation of the traces we apply the traces to tcpreplay which replays the traces with arbitrary speed onto our testing environment.

For the measurements we created a BGP file containing all possible /16 subnets, except the ones containing private, multicast and reserved IP's. This resulted in approximately

56k routes. For all this prefixes we generated a random AS path and made sure that the IP of Laptop1 in figure 11 is set as next hop. We announced these BGP routes to R2 with bgpsimple and waited until all routes were advertised to R1. After all messages were processed by our *supercharged router*, we replayed traffic traces. The traffic traces we used are anonymized real traces without payload from [28]. We removed all packets with IPv6 and private IPv4 destinations from the traces and added a sequence number as payload to identify each packet. The packets are sent from Laptop1 to R1, which will route them either via the virtual switch on Laptop3 or directly to R2. Since we set the IP of Laptop1 as next hop for all announced routes the packets will finally come back to the sender where they will be recorded. The optimization function was turned off for all measurements.

3.2.2 Encountered Problems

As mentioned before we encountered some problems with the setup on real devices:

Buffer overflow While trying to announce 500k prefixes for the first time we had a buffer overflow at the input buffer of our algorithm. The algorithm was not able to handle the update messages in a sufficient small amount of time. To solve this problem we implemented the burst mode (see section 2.4).

Speed of Static Flow Pusher After we solved the buffer overflow problem we found out that the algorithm overloads the Static Flow Pusher part of the REST API by installing to many flow rules in a short time. We had to insert a delay to solve this problem.

Notebook Memory Right from the beginning we had different crashes of Floodlight and ExaBGP. As we found out this was the case because the Laptop run out of main memory space. After we replaced it with a Laptop containing more memory, this issue was resolved.

Floodlight Errors and Packet Loss After we solved the other problems we were able to announce up to 558k routes to the *supercharged router* with arbitrary router FIB sizes. Furthermore, we could send packets to both, prefixes stored in the FIB and prefixes stored in the OVS. But as soon as we replayed the traces, we had multiple errors in Floodlight and sometimes high packet losses. We are still not certain that the observed errors are the only reason for the packet losses since sometimes we also encountered packet loss when all routes are stored in the router's FIB, but it looks like it has a high influence on it. Unfortunately, we had not enough time to resolve this issue.

3.2.3 Delay and Packet Loss Measurements

To measure the delays we created a file containing routes for all possible /16 prefixes, except for the private, reserved and multicast IPs. Then we use bgpsimple to announce the resulting 56k prefixes to R2, which advertised all these routes to our *supercharged router*. We measured the time needed to add each route to the router's FIB or the OVS. As soon as all routes where either announced to R1 or a FlowRule was added to the

OVS's forwarding table, we started to send traffic from Laptop1. As explained above, the packets sent eventually returned to this Laptop. We captured all incoming packets and used this data to measure the delay between sending and receiving for each packet. Additionally, we took the generated log file of the algorithm to determine which prefixes were in the router and which in the OVS. We used this approach to measure the delay and packet loss for different router FIB sizes ranging from 1 (only default route in the FIB) to 100k (all routes in the FIB) to get an impression of how the FIB size affects the delay and number of lost packets.

The graph in figure 12 shows the times needed to announce a new route. For this measurement we used a FIB size of 30k. Each dot in the graph represents the execution time of the *FIB split announce*. The big difference between this measurement and the measurement made in the simulation is that this one includes the time needed to call the APIs of Floodlight and ExaBGP, respectively. The average times to announce a route to the FIB and to the OVS are indicated by the red line. The huge difference between those two devices is due to the delay we had to add between REST API calls.

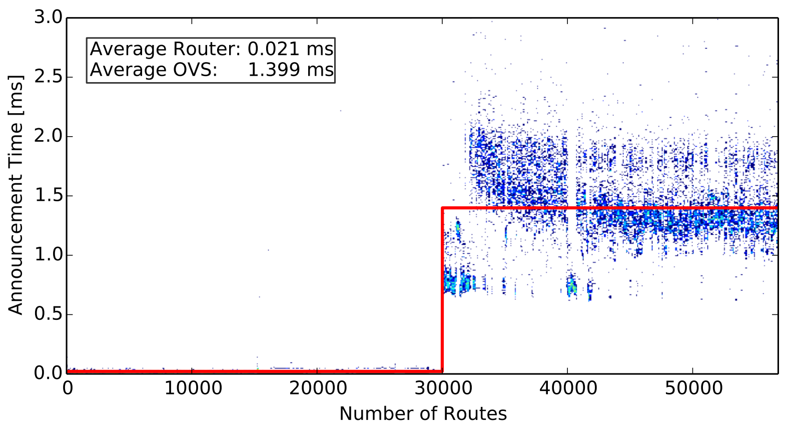


Figure 12: The graph shows the time needed to announce each route. The red line marks the average time needed before and after the FIB is filled.

In all following figures, each dot represents the delay between sending and receiving a single packet, while the red line marks either the average delay or the packet loss. The x-axis shows the time when the packet was received, normalized to the time of the first packet.

Figure 13 shows the delay for packets toward prefixes stored in the FIB compared to the ones stored in the OVS. The measured average delay for packets towards prefixes stored in the FIB is 0.074 ms, while the ones towards prefixes in the OVS is 0.252 ms. This increase of less than 0.2 ms is acceptable since this only affects prefixes stored in the OVS.

The figures 14, 15, 16 and 17 show the measurements for different FIB sizes.

In figure 14 the router's FIB size is bigger than the number of routes, so all routes are stored in the FIB. This situation is similar to a conventional setup without an OVS and controller, except for the fact that the BGP connection is made with ExaBGP. As expected, there is almost no packet loss in this situation (zero in this particular measurement, but in others we lost some single packets).

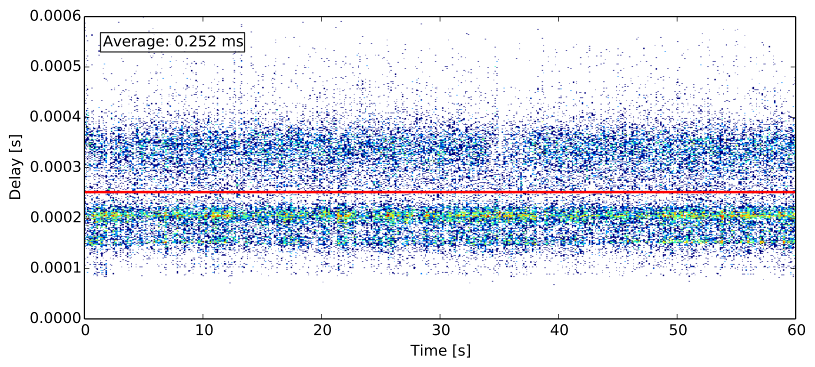


Figure 13: Delay per packet and average delay for prefixes in the OVS (top) and router (bottom).

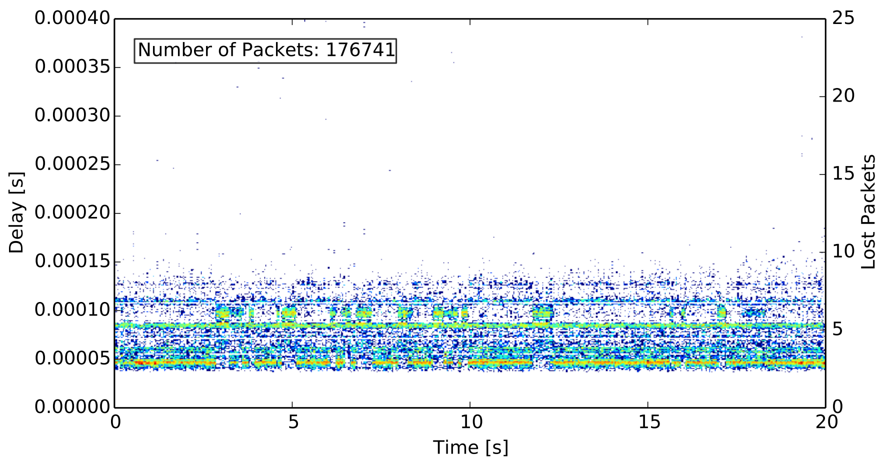


Figure 14: Packet delay and packet loss for the case that all routes are stored in the FIB.

The graph in figure 15 shows the opposite: The FIB size is set to one, so only the necessary default route is stored in the FIB while all advertised routes are stored in the OVS. Here we had the above mentioned errors in the first half of the measurement and the resulting high packet loss.

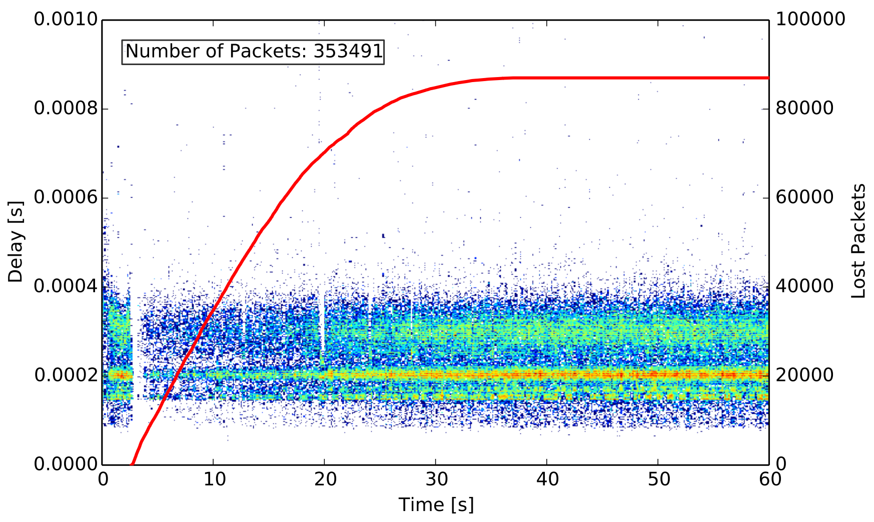


Figure 15: Packet delay and packet loss when all routes are stored in the OVS.

The figures 16 and 17 represent the measurements for 30k and 50k router FIB sizes. In this graphs, the delays and packet losses are separated for the prefixes stored in the OVS (top) and the router (bottom).

The comparison of the different FIB sizes can be found in table 1. We consider only packets which were received in all four measurements to make sure that exactly the same packets are used for the comparison. We use 200k packets of each mesurement for the calculations, and as mentioned before 56k BGP routes were announced.

This values show that our solution has impact on the packet delay, but it has to be considered that we did not optimize the route allocation in these measurements. We believe that this values still are acceptable and will get better as soon as the optimization takes place.

	Router only	50k FIB	30k FIB	OVS only
Packets through OVS	0 %	9 %	27 %	100 %
Max	1.2722 ms	2.2628 ms	0.9980 ms	4.1931 ms
Min	0.0360 ms	0.0360 ms	0.0360 ms	0.0648 ms
Mean	0.0725 ms	0.0922 ms	0.1227 ms	0.2393 ms
Median	0.0710 ms	0.0799 ms	0.0858 ms	0.2179 ms
99th percentile	0.1271 ms	0.4411 ms	0.3910 ms	0.3860 ms
90th percentile	0.1061 ms	0.1352 ms	0.2851 ms	0.3281 ms

Table 1: Comparison of measurement date for different FIB sizes.

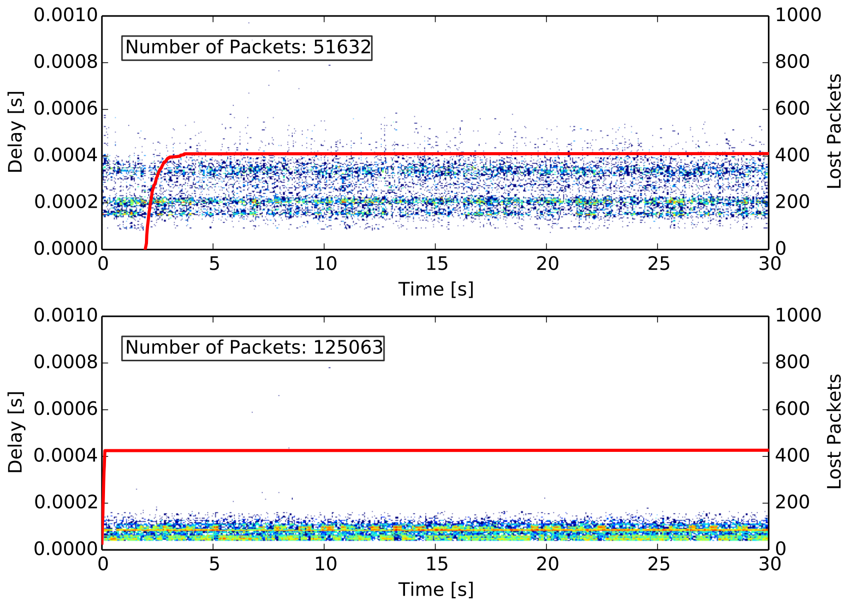


Figure 16: Packet delay and packet loss for routes in the OVS (top) and routes in the FIB (bottom) whit 30k FIB size.

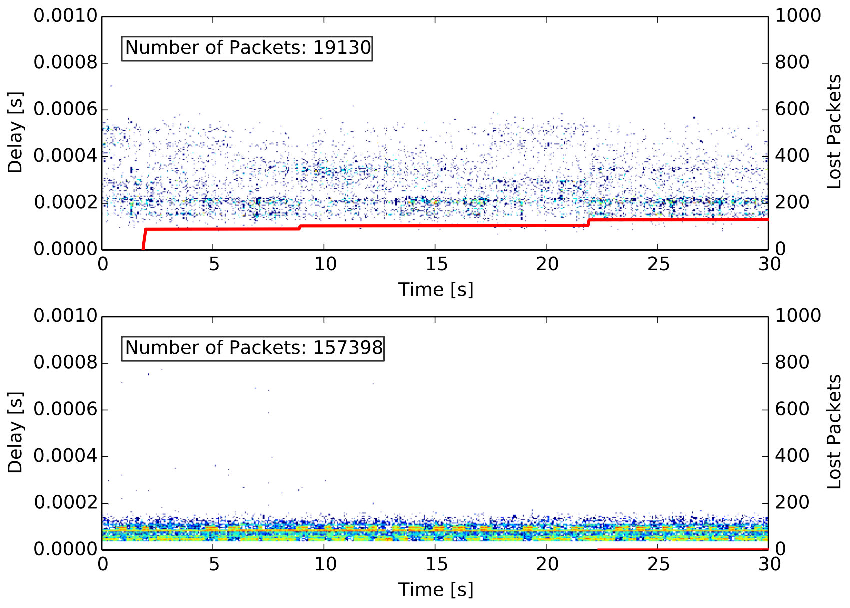


Figure 17: Packet delay and packet loss for routes in the OVS (top) and routes in the FIB (bottom) whit 50k FIB size.

3.3 Discussion

The outcome of the simulation shows that the time needed for the route announcements, as well as the execution time of the optimization are linear to the number of routes advertised by the neighbors. Therefore, with the appropriate hardware, the algorithm can handle today's amount of BGP routes and should also withstand a further increase.

With the performed measurements on the prototype, we showed that our solution generally works on physical devices and the measured delay is acceptable as long as the heavily used prefixes are stored in the router's FIB. Surely, there is still work to do, but the foundation is laid.

4 Outlook

This project lays the foundation to increase an IP routers FIB size with SDN. However, there is still work to do before the *supercharged router* can be used in a productive environment. We show several proposals in this section:

Errors in Floodlight The very first thing which has to be done is surely to solve the last and unsolved problem we encountered in the computer lab. After that is done the missing measurements, for example comparison of average delays before and after the optimization, have to be made.

Separation of ExaBGP from Floodlight and the OVS If ExaBGP runs on an additional server the BGP connection between all neighbors is still provided even if the OVS or Floodlight crash. This would increase the stability of the *supercharged router* and all flows to prefixes stored in the router's FIB would be forwarded correctly. In the actual setup this is not done, but it would be possible with minor changes in the configurations and the algorithm. Moreover, the OVS and Floodlight could also be separated for additional stability.

DoS attack prevention The design proposed in this project is vulnerable to Denial of Service (DoS) attacks. The default rule which we use to forwards all traffic towards prefixes not in the FIB to the OVS could be used to either overload the OVS or cause the optimization function to wrongly swap a lot of routes. An appropriate DoS prevention in the router is indispensable.

Hardware evaluation For our lab setup we simply used a laptop. To use the proposed solution in a productive environment it's necessary to use suitable hardware. We suggest to test it on different hardware to determine which one suits best for this purpose.

5 Conclusion

In this project we propose an approach to increase an IP routers FIB size with SDN to provide a way to handle the increasing amount of BGP routes in the global routing table. We present an algorithm which splits the FIB and stores part of it in an OVS. By using NetFlow data, the algorithm additionally calculates the optimal route split to keep the most used routes in the routers FIB.

The proposed algorithm was evaluated in an offline simulation and the results showed, that the measured execution times were linear to the number of BGP routes. This linearity is essential to withstand the increasing number of BGP routes in today's Internet.

The offline simulation did work well and the results proved the linearity which we tried to achieve, but the physical setup and measurements did not work out as well. Nevertheless, we could solve most of the encountered problems and get some measurements, which showed that our solution works on physical hardware and that the introduced delay is acceptable.

As a closing statement it can be said, that the foundation to increase an IP router's FIB size with SDN is laid, but there is still enough room for improvements.

Bibliography

[1] CIDR REPORT. `http://www.cidr-report.org/`, 2015. [Online; accessed 14-June-2015].

[2] N. Sarrar, S. Uhlig, A. Feldmann, R. Sherwood, and X. Huang. Leveraging zipf's law for traffic offloading. *ACM SIGCOMM Computer Communication Review*, 2012.

[3] M. A. Chang, T. Holterbach, M. Happe, and L. Vanbever. Supercharge me: Boost router convergence with sdn, 2015.

[4] K. Gadkari, M. L. Weikum, D. Massey, and C. Papadopoulos. Pragmatic router fib caching.

[5] X. Zhao, Y. Liu, K. Nam, L. Wang, and B. Zhang. On the aggregatability of router forwarding tables. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9. IEEE, 2010.

[6] Y. Liu, X. Zhao, K. Nam, L. Wang, and B. Zhang. Incremental forwarding table aggregation. In *Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE*, pages 1–6. IEEE, 2010.

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

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

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

[10] Y. Liu, V. Lehman, and L. Wang. Efficient fib caching using minimal non-overlapping prefixes. *Computer Networks*, 2015.

[11] N. Katta, O. Alipourfard, J. Rexford, and D. Walker. Infinite cacheflow in software-defined networks, 2014.

[12] H. Ballani, P. Francis, T. Cao, and J. Wang. Making routers last longer with viaggre. In *NSDI*, volume 9, pages 453–466, 2009.

[13] Open Networking Foundation. Software-defined networking: The new norm for networks. *ONF White Paper*, 2012.

[14] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2), 2008.

[15] J. Hawkinson and T. Bates. Guidelines for creation, selection, and registration of an Autonomous System (AS). RFC 1930 (Best Current Practice), March 1996. Updated by RFCs 6996, 7300.

[16] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271 (Draft Standard), January 2006. Updated by RFCs 6286, 6608, 6793.

[17] B. Claise. Cisco Systems NetFlow Services Export Version 9. RFC 3954 (Informational), October 2004.

[18] Open vSwitch. Open vswitch. `http://openvswitch.org/`, 2014. [Online; accessed 14-June-2015].

[19] Project Floodlight. Floodlight. `http://www.projectfloodlight.org/floodlight/`, 2015. [Online; accessed 14-June-2015].

[20] Floodlight. Rest api. `https://floodlight.atlassian.net/wiki/display/floodlightcontroller/Floodlight+REST+API`, 2015. [Online; accessed 03-July-2015].

[21] Floodlight. Static flow pusher. `https://floodlight.atlassian.net/wiki/display/floodlightcontroller/Static+Flow+Pusher+API+%28New%29`, 2015. [Online; accessed 03-July-2015].

[22] Exa-Networks. Exabgp. `https://github.com/Exa-Networks/exabgp`, 2015. [Online; accessed 15-June-2015].

[23] nfdump. nfdump description. `http://nfdump.sourceforge.net/`, 2014. [Online; accessed 15-June-2015].

[24] Ripe network coordination center, updates 5-july-2015. `http://data.ris.ripe.net/rrc00/`. [Online; accessed 10-July-2015].

[25] bgpsimple. `https://code.google.com/p/bgpsimple/`, 2011. [Online; accessed 03-July-2015].

[26] Tcpdump. `http://www.tcpdump.org/tcpdump_man.html`, 2014. [Online; accessed 03-July-2015].

[27] Tcpreplay. `http://tcpreplay.synfin.net/`, 2014. [Online; accessed 03-July-2015].

[28] The caida ucsd anonymized internet traces 2015 - equinix-chicago. `http://www.caida.org/data/passive/passive_2015_dataset.xml`, 2015. [Online; accessed 09-July-2015].