7-24-2013

Reducing Router Forwarding Table Size Using Aggregation and Caching

Yaoqing Liu

REDUCING ROUTER FORWARDING TABLE SIZE USING AGGREGATION AND CACHING

by

Yaoqing Liu

A Dissertation

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy of Science

Major: Computer Science

The University of Memphis

August 2013

Abstract

Liu, Yaoqing. Ph.D. The University of Memphis. Reducing router forwarding table size using aggregation and caching. Major professor: Dr. Lan Wang.

The fast growth of global routing table size has been causing concerns that the Forwarding Information Base (FIB) will not be able to fit in existing routers' expensive line-card memory, and upgrades will lead to a higher cost for network operators and customers. FIB Aggregation, a technique that merges multiple FIB entries into one, is probably the most practical solution since it is a software solution local to a router, and does not require any changes to routing protocols or network operations. While previous work on FIB aggregation mostly focuses on reducing table size, this work focuses on algorithms that can update compressed FIBs quickly and incrementally. Quick updates are critical to routers because they have very limited time to process routing updates without impacting packet delivery performance. We have designed three algorithms: FIFA-S for the smallest table size, FIFA-T for the shortest running time, and FIFA-H for both small tables and short running time, and operators can use the one best suited to their needs. These algorithms significantly improve over existing work in terms of reducing routers' computation overhead and limiting impact on the forwarding plane while maintaining a good compression ratio.

Another potential solution is to install only the most popular FIB entries into the fast memory (*i.e.*, an *FIB cache*), while storing the complete FIB in slow memory. In this paper, we propose an effective FIB caching scheme that achieves a considerably higher hit ratio than previous approaches while preventing the

cache-hiding problem. Our experimental results using data traffic from a regional network show that with only 20K prefixes in the cache (5.36% of the actual FIB size), the hit ratio of our scheme is higher than 99.95%. Our scheme can also efficiently handle cache misses, cache replacement and routing updates.

Acknowledgement

I thank my advisor, Dr. Lan Wang, who played a key role during my PhD studies at the University of Memphis. She guided and directed me for all of my research projects. Also she provided me with a lot of post-graduate career help.

I thank all of my family members, especially my beloved wife Yanru Qiao and two-and-half-year-old son Timothy Liu. Without their support and encouragement, my dissertation must look different.

I extend my thanks to all of the committee members, who offered valuable comments or suggestions to the dissertation.

I also extend gratitude to my surrounding friends, such as Vincent Scott Lehman, Danesha Winfrey, etc., who helped the proofreading, and Yongmei Wang and Barbara Mullins Nelson who taught me both useful career advice and skills for job interviews.

Finally, I reserve this space to appreciate those I have left out but would acknowledge, and you, the readers of this dissertation.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1 Routing Scalability Issue

The global routing table size has been increasing faster than ever in a super-linear trend, due to a variety of factors such as an increasing number of edge networks, increased use of multihoming, and finer-grained traffic engineering practices [57].

This is the so-called routing scalability problem, which has raised concerns in both industry and research communities, as documented in the report from the IAB Workshop on Routing and Addressing. Several solutions have been proposed under the IRTF RRG [6] and IETF GROW working groups [5]. A direct consequence of this problem is the rapid growth of the forwarding table (FIB: Forwarding Information Base) size. Although both trends are disturbing, ISPs are more concerned about the FIB size [36], because the FIB memory in line cards costs much more than the memory in router processors as the former needs to support much higher lookup speed at the line rate (*e.g.*, hundreds of millions of packets per second or higher). Moreover, as the size of FIB memory increases, the FIB lookup time may also increase [76] and the line card may become more power-hungry as well [57]. Once the FIB becomes so large that it can no longer fit in the fast memory of routers' line cards, ISPs will have to upgrade their line cards, eventually making Internet services more expensive. To address the root cause of the

scalability problem, fundamental changes to the Internet routing architecture and protocols are called for. However, deploying architectural changes is likely to take a long time, as illustrated by past examples like IPv6. While architectural changes may benefit the Internet in the long run, short-term solutions are needed as the problem is serious and imminent. In particular, ISPs urgently need to reduce their forwarding table size. Forwarding tables are derived from routing tables and router configurations, thus their size increases as routing tables grow. However, forwarding tables use high performance memory that is more expensive and more difficult to scale than the memory used to hold routing tables. Therefore, their size is a more immediate concern to ISPs and vendors. While a number of solutions (*e.g.*, [20] [42] [41] [45] [31] [34] [21]) have been proposed to solve the routing table scalability problem by changing the routing architecture in the long run, ISPs need practical solutions soon, and FIB aggregation is considered one of the most practical solutions [83]. This work investigates two solutions to reduce the routing forwarding table size: FIB aggregation and FIB caching.

2 Solution 1: FIB Aggregation

FIB aggregation, exploring the feasibility of a purely local solution, combines multiple entries in the forwarding table without changing the next-hops for data forwarding. This approach is particularly appealing because it can be done by a software upgrade on a router, therefore, its impact is limited within the router. It does not require changes to routing protocols or router hardware, nor does it affect multi-homing, traffic engineering, or other network-wide operations. It is important to note that FIB aggregation is not a replacement for long-term architectural solutions because it does not address the root causes of the routing scalability problem. Instead, FIB aggregation is a local solution that can be quickly

implemented and deployed in the short-term, and it can co-exist and complement architectural solutions in the long run. The idea of FIB aggregation is rather intuitive, but to our best knowledge, no study has systematically evaluated its potential benefits or costs. FIB aggregation is an opportunistic technique-its effectiveness depends on what prefixes are present in the table, how many of them can be numerically represented by a single prefix, and how many of them share the same next-hop. The benefits of FIB aggregation come with certain costs, such as extra CPU cycles. The costs also depend on the actual aggregation algorithms and how routing changes are handled to update the aggregated forwarding table. A thorough understanding of FIB aggregation is needed in order to decide whether it is a viable solution.

FIB aggregation reduces FIB size by combining entries whose prefixes are numerically aggregatable and whose next-hops are the same. It is a software solution that can be applied to a single router without upgrading the hardware, changing the control plane, or affecting packets' forwarding paths. Thus it can be deployed incrementally and selectively in a network at operators' discretion. One of the fundamental tradeoffs in FIB aggregation is between aggregated table size and computational overhead. Spending too many CPU cycles aggregating the table will delay it from being downloaded into the line cards, which may lead to packet loss or incorrect forwarding. Existing work (*e.g.*, [29] [82] [53] [79]) has demonstrated that FIB aggregation can reduce table size by as much as 70% with moderate computation, but these efforts have not focused on reducing routers' overhead in all aspects.

The most challenging problem in FIB aggregation is to quickly apply updates to the already aggregated table and still maintain a good compression ratio. When a router receives a routing update, it has a very limited amount of time to process the update and install the new FIB. When the FIB is already aggregated, one routing

table change may lead to updating multiple FIB entries, because it may change the aggregatability of those entries. In some cases, there can be thousands or even tens of thousands of FIB entries to be updated, even if there is only a single routing table change. Therefore in this work, we focus on reducing FIB aggregation's overhead in the following aspects: (1) reducing the overall time of processing a stream of updates; (2) speeding up the process to re-aggregate an entire FIB, if a scheme requires such re-aggregations; and (3) reducing the average and maximum number of FIB changes caused by any individual routing table change, so as to reduce the time it takes to push those changes to the line card.

To this end, we have designed three algorithms: FIFA-S for the smallest table size, FIFA-T for the shortest running time, and FIFA-H for both small tables and short running time. They take advantage of some intrinsic properties of an aggregated FIB trie to speed up the incremental update process. Among them, FIFA-S and FIFA-H do not need to run full re-aggregations, and FIFA-T performs fast re-aggregation on the existing aggregated trie. Moreover, they use a prioritized set of next-hop selection rules to improve the stability of the aggregated FIB thus reducing the number of FIB changes per routing table change. Our evaluation shows that they outperform state-of-art algorithms in both speed and FIB stability.

3 Solution 2: FIB Caching

As mentioned in Solution 1, modifying the current routing architecture and protocols seems to be the best long-term solution to these problems [49]. However, such proposals may take a long time to deploy due to the high costs associated with them. Meanwhile, ISPs cannot afford to frequently upgrade all of their routers. Zhao *et al.* investigated various possibilities to mitigate the routing scalability issue and concluded that FIB size reduction would be the most promising solution from

an operator's view [83]. Two main solutions can lead to FIB size reduction, FIB aggregation and FIB caching. FIB aggregation is to aggregate a large FIB table into a smaller one with the same forwarding results. There has been a number of FIB aggregation algorithms proposed in the last few years ([29], [11], [82], [53], [79]). The aggregation results show that the FIB size can be reduced to 1/3 of the original table size, at most. According to [79], even with the state-of-the-art FIB lookup algorithm, called Tree Bit Map [30], the actual saved memory is half of the original FIB memory and reducing more than this seems impossible. Therefore, FIB caching is another promising solution to reduce the routing forwarding table size. A router stores the routes for the known destination prefixes in a routing table, known as a Routing Information Base (RIB). The path selection algorithm uses the RIB to find out the best route for a particular destination and pushes it to the Forwarding Information Base (FIB), which is responsible for packet forwarding. All alternative routes remain in RIB and can be used when the best route is unavailable. In recent years, a tremendous growth in RIB, and consequently of FIB, has been observed, which is concerning for ISPs as its implications include high management cost, inefficient forwarding, and more power consumption, just to name a few. To scale well with the increasing FIB size, a naive solution is to add more memory to the routers. However, firstly, it is challenging to meet the continuously changing memory requirement and secondly, as FIB memory is expensive, upgrading the memory of all routers of an AS is not compelling for ISPs.

One approach to reducing the impact of large FIBs is to use high-speed memory as a *cache* to store the most popular routes [19, 46, 50, 81] while storing the full FIB in lower-cost memory. The feasibility of this approach, which we call *FIB Caching*, depends on how much locality is in the network traffic. In fact, previous studies [37, 46, 67, 81] have shown that a small number of popular prefixes contribute to most of the observed traffic. The data trace from a regional ISP used in our

evaluation also supports this observation. As such, a FIB cache needs to store only a small set of popular prefixes thus saving a router's high-speed memory, increasing lookup speed, and reducing power consumption.

Although caching has been studied extensively in general, *FIB caching* has its unique set of issues. First, network links forward a huge number of packets every second, which means even a 1% miss ratio could lead to millions of lookups per second in *slow* memory. To minimize this problem, an effective FIB caching scheme must achieve an extremely high hit ratio with a modest cache size. Second, the cache miss problem is especially serious when a router starts with an empty cache, so a good scheme needs to quickly and effectively fill the cache even without prior traffic information. Third, Internet forwarding uses longest-prefix matches rather than exact matches. If designed poorly, a FIB caching scheme may cause a *cache-hiding* problem, where a packet's longest-prefix match in the cache differs from that in the full FIB; thus the packet will be forwarded to the wrong next-hop (Section 1). To prevent this problem, prefixes for the cache need to be carefully selected from the full FIB or dynamically generated. Finally, prefixes and routes change from time to time, therefore, any practical FIB caching scheme needs to handle these changes efficiently without causing the cache-hiding problem.

We propose a FIB caching scheme that selects and generates a minimal number of *non-overlapping* prefixes for the cache. Because the cached prefixes do not cover any longer prefixes in the full FIB, we do not have the cache-hiding problem. Based on this caching model, we developed a FIB caching update algorithm to systematically handle cache misses, cache replacements and routing updates. Our experimental results using data traffic from a regional network show that, for a routing table of 372K prefixes, our scheme achieves a hit ratio higher than 99.95% using a cache size of 20K prefixes (5.36% of the full FIB size) and outperforms alternative proposals in term of hit ratio. In addition, we fill the *initial* cache with

the *shortest* non-overlapping prefixes generated from the full FIB which significantly increases the hit ratio for the initial traffic. Our evaluation results show that the initialized cache has a hit ratio of 85% for the first 100 packets compared to 65% for an uninitialized cache.

The remainder of the dissertation is organized as follows. Chapter 2 gives background and related works on FIB aggregation and caching; Chapter 3 presents the design, evaluation and implementation of FIB aggregation algorithms; Chapter 4 presents the design, evaluation and implementation of FIB caching architecture; and Chapter 5 concludes the dissertation.

Chapter 2

Background

1 Background

1.1 Router

Routers [14] glue every small network into the global Internet and various types are used at every level. For example, homes and small businesses utilize home routers or access networks to connect with their Internet Service Providers (ISPs); enterprise networks or campus networks utilize routers to connect tens, even thousands of computers together; backbone networks utilize routers to link ISPs and enterprise networks together which are sometimes called Wide Area Networks. The basic functionalities for a router are route collection, route selection, and packet processing. Packet processing includes packet receiving, packet buffering, packet route searching, and packet forwarding [61, 72, 77, 78, 27]. In order to implement all the functionalities, a router, especially for a backbone network, may contain different types of memory, such as DRAM (Dynamic Random Access Memory), SRAM (Static Random Access Memory) and TCAM (Ternary Content Addressable Memory) [3]. DRAM is cheap but slow and can be used for route collection and selection; SRAM is fast but expensive and can be used for packet buffering; TCAM is the fastest but also the most expensive and generally can be used for packet route searching and forwarding [15].

1.2 Routing Table and Forwarding Table

There are two types of tables used by routers: *Routing Information Base (RIB)* for routing and *Forwarding Information Base (FIB)* for forwarding. RIB is stored in the main memory of a route processor. The route processor receives and processes routing update messages and runs routing protocols, *e.g.*, OSPF [59] and BGP [65], to compute the RIB. Each RIB entry contains the destination IP prefix and associated route information. For example, BGP maintains full AS path and many other attributes for each prefix in RIB. FIB is derived from RIB and router configurations and is stored in line cards, which forward data packets. Therefore, FIB usually uses high performance memory, which is more expensive and more difficult to scale. For each destination IP prefix, the FIB has an entry to store the next-hop IP, next-hop MAC address and outgoing interface for fast data forwarding. Figure 2.1 illustrates these different components in a router.

Routers use their routing protocols, e.g. OSPF [59] and BGP [65], to compute their routing table or *Routing Information Base (RIB)*. Each entry in a RIB includes an address prefix, e.g. 3.0.0.0/8, and the corresponding route information, e.g. the next-hop's IP address. RIB is usually stored in the memory of a route processor. While the route processor is responsible for computing routing information, routers use dedicated processors in *line cards* to forward packets. Each line card has its own or shared forwarding table (also called the *Forwarding Information Base or FIB*) which is derived from the RIB and router configurations. For each address prefix, the FIB contains the outgoing interface, the next-hop's IP address, and the next-hop's MAC address for fast forwarding. Whenever a change to the RIB results in a different next-hop for an address prefix, the FIB has to be updated accordingly. Figure 2.1 illustrates these different components in a router.

Despite growth constraints such as strict address allocation policies [57], the

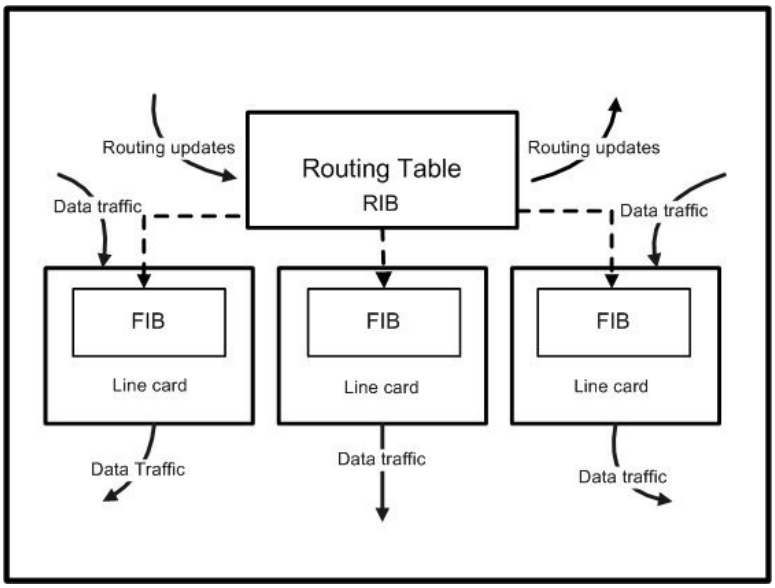


Figure 2.1: RIB and FIB

routing tables in the default free zone (DFZ) have been growing at an alarming rate in recent years. Currently, a DFZ router stores hundreds of thousands of routes or even a million in tier-1 ISPs. This is due in part to the sheer growth of the Internet, as well as lack of aggregation. When a customer network multi-homes to multiple providers for resilient Internet connectivity, the customer's address prefix(es) must be visible in the global routing table in order to be reachable through any of its providers, thus breaking down provider-based aggregation [24]. Traffic engineering is another contributing factor. For example, a network may try to influence the paths of specific incoming traffic flows by splitting its prefix into several longer ones and injecting them at different network attachment points. Splitting prefixes is also used as a defense mechanism against prefix hijacking, since hijacking longer prefixes is less effective than hijacking shorter prefixes due to router's longest-match routing lookup. Growing table size leads to increasing FIB tables, RIB tables, and routing churns. Among these problems, ISPs and vendors are more concerned about the FIB size than RIB size, because it is more difficult to scale up the memory in line

cards than in route processors [36]. Due to this, the price of a line card in a high performance backbone router is very expensive, the price of a state-of-the-art router line card has been over 200, 000 dollars per piece.

The conventional way of reducing routing table size is to aggregate the RIB, which will also reduce FIB size. However, RIB aggregation has very limited adoption in the Internet. At a prefix's origin network, there is little incentive to aggregate the prefix, because the gain of aggregating a small number of self-originated prefixes does not make much difference to the table size. At the same time, the origin network actually has incentives, such as multi-homing and traffic engineering, to split the prefix. At a remote site, aggregation opportunity is limited since two prefixes must have the same *path attributes* in order to be aggregated in RIB. Otherwise, their path information will be lost and protocol functions may be affected. Forcing aggregation of prefixes that have different paths would also defeat multi-homing and traffic engineering intended by the prefix origin networks.

Definition 1. *A FIB (F) contains a set of forwarding entries,* i.e., $F = \{(p, h)\}$, *where h is a set of next-hops for forwarding packets to any IP address in prefix p.*

Definition 2. *Given an IP address d and a FIB F, let $LPM(F, d)$ denote d's Longest Prefix Match, an address prefix $p = a/l$ in F is the* ***Longest Prefix Match (LPM)*** *for d,* i.e., $p = LPM(F, d)$, *if and only if the following conditions hold: (1) $d = a\{0, 1\}^*$, and (2) for any address prefix $p' = a'/l' \neq p$ in F, if $d = a'\{0, 1\}^*$, then $l' < l$. and $nexthop(F, p)$ denote the next-hops for prefix p. We define $nexthop(F, d) = nexthop(F, LPM(F, d))$. It is possible that d does not have any match in the FIB, i.e., $LPM(F, d) = NULL$, and packets destined to d will be dropped.*

As an example, Table 2.1(a) shows a FIB F with five entries. For address 141.225.48.7, this address matches 141.225.0.0/16, 141.225.32.0/19 and

Table 2.1: FIB entries before and after aggregation

(a) Original FIB Entries

Label	Prefix	Next-hop
A	141.225.0.0/16	1
B	141.225.64.0/18	1
C	141.225.32.0/19	1
D	141.225.96.0/19	2
E	141.225.48.0/20	2

(b) Aggregated FIB Entries

Label	Prefix	Next-hop
A	141.225.0.0/16	1
D	141.225.96.0/19	2
E	141.225.48.0/20	2

141.225.48.0/20, among which 141.225.48.0/20 is the longest prefix match, i.e., $LPM(F, 141.225.48.7) = 141.225.48.0/20$, and $nexthop(F, 141.225.48.7) = nexthop(F, 141.225.48.0/20) = \{2\}$.

1.3 FIB Aggregation and Forwarding Correctness

FIB aggregation eliminates and aggregates entries in a FIB based on the next-hop router information while ensuring forwarding correctness. For example, it can remove prefix $P1$ from the FIB if its super-prefix $P2$ uses the same next-hop as $P1$. It may also introduce a new entry to the FIB after removing multiple entries that share the same next-hop. FIB aggregation may be more effective than RIB aggregation since it only requires prefixes to have the same next-hop in order to be aggregated. For example, considering that a Los Angeles router connects to a Tokyo router, which in turn connects to a Beijing router and a Shanghai router. The Los Angeles router may reach prefixes announced by China Telecom via different paths, some via Beijing and some via Shanghai. However, in its FIB, most of these prefixes take the Tokyo router as the next-hop, making them aggregatable.

The effectiveness of FIB aggregation depends on how prefixes are distributed over next-hop routers. Generally speaking, the fewer neighbors a router has, the better aggregation it may achieve. In the extreme case that all prefixes share the same single next-hop, aggregation is maximized. According to Li *et al.* [47],

although some routers have high degrees up to a few hundreds, most connections are with their end-customers, which represent only a small percentage of the address space. The routers still use a small number of transit neighbors to reach most address prefixes. Besides sharing the same next-hop, prefixes also need to be numerically aggregatable. This is possible due to two factors. First, in IP address allocation, large blocks of Internet addresses are first allocated to Regional Internet Registries and then they further allocate the addresses to networks within the same region. Thus prefixes announced out of the same regions tend to be numerically aggregatable. Second, for prefixes split for traffic engineering or other purposes, a router near the origin network is likely to take different next-hops, but a router further away from the origin network is more likely to have the same next-hop towards these numerically aggregatable prefixes.

Therefore, although FIB aggregation is opportunistic and the aggregation degree varies from router to router, there are inherent properties of the Internet that can make FIB aggregation effective. If FIB aggregation is indeed effective in reducing table size, its most appealing feature is that the impact is limited within a router's data plane. It does not change any routing protocols, or any router's routing decisions. Data traffic still flows on the same router paths. Therefore, it can co-exist with almost any new routing protocols, including long-term architectural solutions to the routing scalability problem.

In a FIB, there are hundreds of thousands of prefixes but only, at most, thousands of next-hops as shown in Table 2.1(a). That means many prefixes share a common next-hop. This fact results in the possibility to do FIB aggregation without affecting the original forwarding behavior. Table 2.1(b) illustrates the prefixes and next-hops after aggregation with the same forwarding behavior as that before aggregation. FIB aggregation is to aggregate a FIB into one with fewer number of entries while ensuring "forwarding correctness", *i.e.*, the aggregated FIB should not

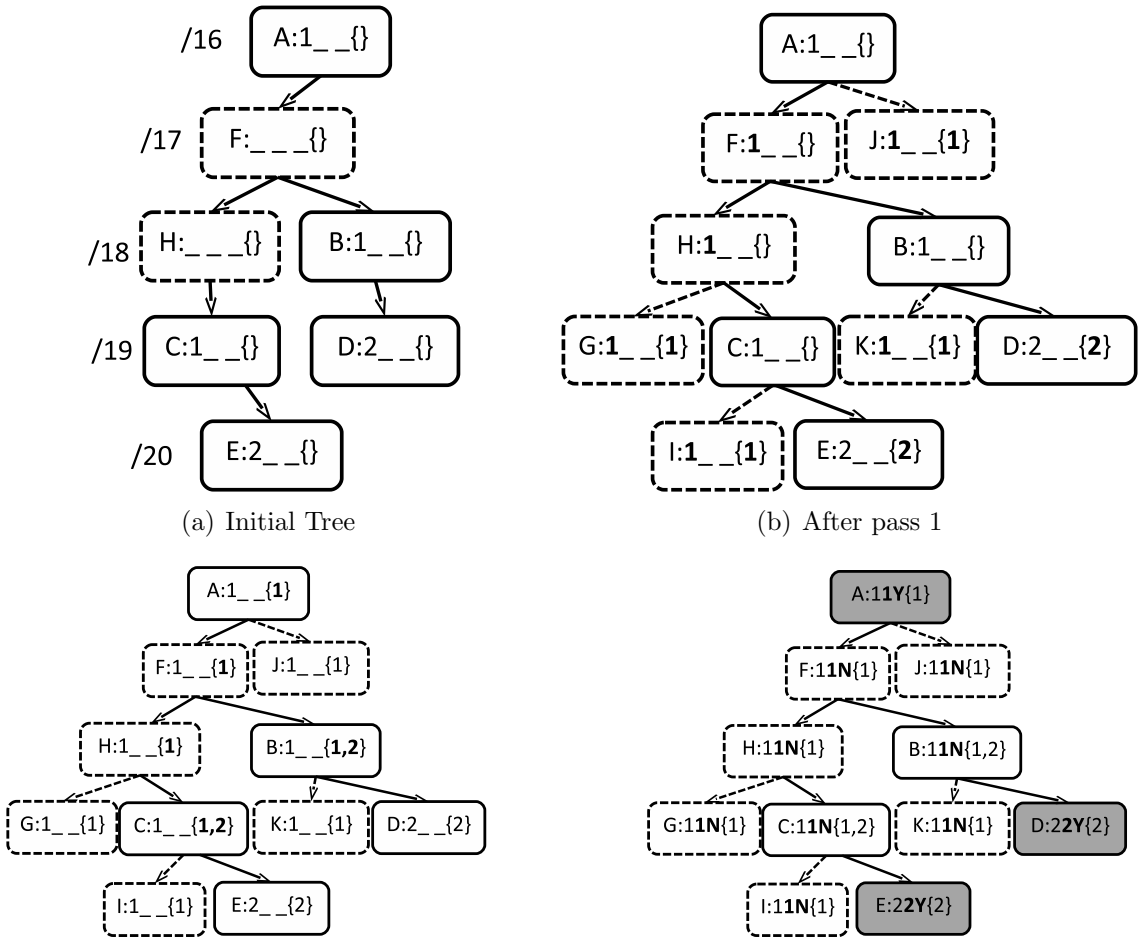
change the next-hops that packets take to reach their destinations. All the FIB aggregation algorithms proposed in this paper satisfy *strong forwarding correctness* as defined below. Note that even if two algorithms satisfy the same type of forwarding correctness, they may reduce a FIB into different sizes depending on what aggregation opportunities they exploit.

Definition 3. *Given a FIB F, another FIB F' satisfies* ***Strong Forwarding Correctness*** *with respect to F if and only if the following conditions hold: (1) any non-routable address in F will remain non-routable in F',* i.e., *if $LPM(F, d) = NULL$, then $LPM(F', d) = NULL$; (2) the next-hop of any routable address in F will remain the same in F',* i.e., *if $LPM(F, d) \neq NULL$, $nexthop(F', d) = nexthop(F, d)$. If only the second condition holds, we say that F' satisfies* ***Weak Forwarding Correctness*** *with respect to F (this means a non-routable address in F can become routable in F').*

In the simplest case, when several consecutive prefixes share a common next-hop, they can be combined into a shorter prefix with the same next-hop. Another simple case is when a prefix and its nearest ancestor prefix share the same next-hop, this prefix can be removed from the FIB. In both cases, the longest prefix match will return the shorter prefix, but the returned next-hop will still be correct. The second case is illustrated in Table 4.1(a), where the entries B and C can be removed from the original FIB – they share the same next-hop as the entry A, and A's prefix 141.225.0.0/16 is their nearest ancestor prefix. There are more complex cases where aggregation can be applied.

1.4 Optimal Routing Table Constructor (ORTC)

Our algorithms are based on the Optimal Routing Table Constructor (ORTC) [29], a one-time aggregation algorithm that minimizes the FIB size with strong



(a) Initial Tree (b) After pass 1

(c) After pass 2 (d) After pass 3

Figure 2.2: ORTC Aggregation Algorithm. There are four fields for each node from left to right: original next-hop, selected next-hop, FIB status (Y: IN_FIB, N: NON_FIB), and next-hop set. A bold font denotes a field updated in the current step. A solid rectangle denotes a *real* node from the unaggregated FIB. A dashed rectangle denotes an auxiliary node generated either as a glue node or for optimization purposes. A grey node denotes a node with IN_FIB status.

forwarding correctness. The basic ORTC algorithm uses a binary tree to store FIB entries and traverses the tree three times to produce the aggregated FIB. As we will show in Section 1.1, ORTC can be implemented using a patricia trie [9] with two tree traversals. However, for ease of illustration, we describe the basic ORTC algorithm using a binary tree and three passes.

We use the FIB in Table 2.1(a) as our example. Figure 2.2(a) shows the initial binary tree with seven nodes. Five of the nodes, A, B, C, D, and E, correspond to the FIB entries in Table 2.1(a). We call them "real" nodes, while the other two nodes, F and H, are called "auxiliary" nodes. Note that some of the nominal nodes may appear in the aggregated FIB, while some of the real nodes may not.

The first pass is a depth-first traversal in pre-order to normalize the tree, so that all the nodes have zero or two children. The expanded nodes have the same next-hops as their nearest ancestors that are real nodes. Figure 2.2(b) depicts the process for pass 1. Node G, I, J and K are the expanded leaf nodes, and they have the same next-hops as their nearest real ancestors A, C, A, and B, respectively.

The second pass is a depth-first traversal in post-order to merge next-hops, in which two children merge their next-hop sets to form their parent's next-hop set. If the two children have one or more common next-hops, the merging uses an intersection operation, otherwise, it uses a union operation. Figure 2.2(c) depicts the merging process. For example, E and I have no common next-hops, so their parent C's next-hop set is {1,2}, the union of {1} and {2}. Another example is H, whose next-hop set {1} is the intersection of C's next-hop set {1,2} and G's next-hop set {1}.

The third pass is a depth-first traversal in pre-order to select each node's next-hop and form the aggregated FIB. The root node's next-hop is randomly selected from its next-hop set (the original next-hop may be preferred for stability). Starting from the tree root, first pick any next-hop from the merged next-hop set as

the selected next-hop of the root. Then execute the following procedure recursively: from then on, if a node's selected next-hop h appears in its child's next-hop set, then the child should have h as its selected next-hop, so that the child will not be loaded into the FIB. Otherwise, the child's next-hop is randomly selected from its next-hop set, and the child will be loaded into the FIB.

Figure 2.2(d) shows the results after pass three. Root A has 1 as its selected next-hop. Since its children F and H have 1 in their next-hop set, they also have 1 as their selected next-hops and, as such, they will not appear in the aggregated FIB. On the other hand, D's selected next-hop (2) is different from that of its parent B (1), so it must be put into the aggregated FIB. Table 2.1(b) shows the final prefixes and their next-hops.

2 Related Work

2.1 Aggregation Algorithms

Optimal Routing Table Constructor(ORTC) [29] was proposed by Draves *et al.* 1997. It aggregates a routing table into its optimal size using three passes over a binary tree, while maintaining strong forwarding correctness. However, this algorithm does not include any update handling mechanism. Moreover, it uses a binary tree as the data structure.

Zhao *et al.* proposed four aggregation algorithms (Level1 - Level4) [82]. Level1 and Level2 algorithms maintain strong forwarding correctness, but they do not optimize the FIB size. Level3 and Level4 achieve weak forwarding correctness by introducing extra routable space. Also, it needs to do a Full Tree Re-aggregation when a threshold is reached.

In 2009, Karpilovsky proposed an incremental FIB update algorithm [43] based on ORTC. This algorithm is similar to FIFA-S. However, it needs three passes to

handle an update, and it normalizes all affected ancestors of an updated node, both of which introduce considerable computational overhead.

In 2010, we proposed two incremental FIB aggregation algorithms [53] using a Patricia Trie data structure [9] based on ORTC. We showed that FIFA-S and FIFA-T outperform these algorithms in Section 2. The optimal size update scheme will always keeps optimal FIB size, but needs a full subtree aggregation for each update, and the process is time consuming. The minimal time update algorithm does a full tree re-aggregation by destroying the old aggregated FIB tree and rebuilding a new FIB tree after the threshold has been reached. The Full Tree Aggregation will trigger large scale number of FIB downloads and very heavy FIB burst to FIB, the detailed results comparison has been illustrated in Section 2.

Another very relevant work is SMALTA [79], in which Uzmi *et al.* implemented a different update handling algorithm based on ORTC. The algorithm is very similar to our original minimal time update algorithm in [53]. The main differences from ours are that they utilize a binary tree rather than a Patricia Trie and update only affected nodes without optimizing the subtree rooted at the updated node. Also, after a certain number of updates (every a few hours), they need to do a Full Tree Re-aggregation to keep FIB table size small and FIB burst light.

Li *et al.* proposed an FIB aggregation scheme with multiple selectable next-hops [48], which is geared toward FIBs with multiple selectable next-hops for each prefix. The scheme can potentially introduce path stretch issue.

FIB aggregation can reduce the FIB size by aggregating a large FIB table into a smaller one with the same forwarding results. There has been a number of FIB aggregation algorithms [29, 82, 53, 79]; their results show that the FIB size can be reduced to at most 30% of the original table size. FIB caching is complementary to FIB aggregation. In fact, the full FIB can be aggregated first and then serve as the basis for caching, which can further reduce the required cache size.

The Virtual Aggregation (VA) scheme [34] tries to install some virtual prefixes which are shorter than real prefixes, such as /6, /7 and /8, to legacy routers to control FIB size growth. It can reduce the FIB size on most routers while the routers that announce the virtual prefixes still need to maintain many more specific prefixes. Our FIB caching scheme can be applied to those routers with a larger FIB size in a network deploying VA.

Simple Virtual Aggregation [64] installs virtual prefixes which are shorter than real prefixes, such as /6, to legacy routers to control FIB size growth. It can reduce the FIB size on most routers, while the routers that announce the virtual prefixes still need to maintain many specific prefixes.

2.2 Compact Data Structures

Regarding more broad concerns about FIB compression, a heavily researched area was to find efficient FIB representations with compact data structures. Trie-based data structures were proposed, such as Patricia Tries [69], Multi-bit Tries [25, 40, 73], Path-compact tries [62], Lulea [28], Tree Bitmaps [22, 30, 71]. Other approaches include theoretical research on compressed data structures [68, 32, 33, 39, 54, 60, 63, 84], labeled tree folding into DAG [44, 23, 26, 70, 74, 66], hash-based compression [80], dynamic pipelining [38], CAMs [55] and so forth. Our FIB aggregation work could combine with all of these works to achieve a smaller usage of line card memories, which may further reduce the costs that ISPs have to pay to replace or upgrade their equipment.

2.3 FIB Caching

Liu proposed Routing Prefix Caching for network processors [50], which employs three prefix expansion methods, NPE, PPTE and CPTE. These solutions can eliminate the inter-dependencies between prefixes in the cache, but they will either

increase the FIB size considerably or have a high miss ratio. CPTE expands the prefix tree into a complete binary tree, in which there are either two children, or no child for a node; this method increases the binary tree size dramatically. NPE does not do prefix expansion and any parent prefix in the cache will be marked as non-cachable. The cache missed IP addresses will be installed in the cache, this method increases the cache miss ratio. PPTE only expands the tree on the first level and other nodes follow the same way as NPE. Then the cache missed IP address will be installed in the cache for other nodes. The advantage of the proposed solution is that they remove inter-dependent prefixes issue if there is a sub-prefix for a super-prefix. However, expanding a prefix tree to a complete binary tree will increase the size of routing table significantly; using non or partial binary tree expansion will cause a high cache missing ratio and a large FIB cache by using IP addresses as the replacement elements.

Akhbarizadeh *et al.* proposed RRC-ME [19]. This solution can also solve the cache-hiding problem through using disjoint prefixes, but it has significant update handling overhead, especially in the worst cases. RRC-ME has a very similar mechanism to our proposal. First, they expand the tree partially based on the destination IP address and only cache disjoint popular prefixes to save memory. However, they use a binary tree structure to generate prefixes, simply pick i+1 bits of the address as prefix (i is the bit length of last visited node). Moreover, in absence of control plane information, any prefix to be updated needs to search the cache several times to find the matching prefix and perform corresponding operations. Furthermore, a withdrawal prefix may result in the deletion of all of its children in the cache. Our proposal uses both control and data plane information, and thus has two sets of schemes to handle both normal update messages and update messages from the cache.

Kim *et al.* proposed route caching using flat and uniform prefixes of 24 bits

long [46]. It can achieve fast access speeds using a flat hash data structure for lookup. However, this approach leads to prefix fragmentation and thus has a lower hit ratio than our approach as shown in our evaluation results. Moreover, no systematic update handling scheme was present in the work.

Chapter 3

FIB Aggregation

1 Design

We aim to develop FIB aggregation algorithms that are practical to use in a real production network. First, they should reduce the FIB size sufficiently to postpone the upgrading of FIB memory in line cards by several years. Second, they should handle route changes quickly as a router may need to handle a large number of routing changes during routing convergence. Third, they should not incur a large number of FIB changes per routing update. According to Francois *et al.* [35], the time required to update a FIB entry in a real router is about $100\mu s$. Since one route change may result in multiple FIB changes on an aggregated FIB, it would be desirable to minimize such FIB changes. Finally, we would like to maintain *strong forwarding correctness* (see Section 1) to avoid potential looping problems associated with weak forwarding correctness.

When a router starts up, FIFA uses our improved version of ORTC ([29] and Section 1) to build the initial aggregated FIB. When a new routing update arrives, the routing protocol will first update the RIB and then FIFA will apply each resulting route change to the aggregated FIB, which may generate one or more FIB changes. FIFA then installs these FIB changes in the line card. Figure 4.3 illustrates this process.

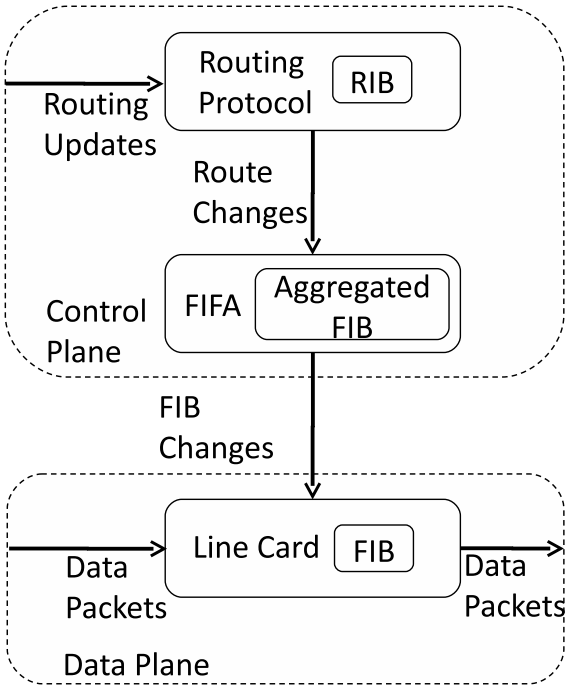


Figure 3.1: Relationship between FIFA and other router components

FIFA is composed of three algorithms: FIFA-S, FIFA-T and FIFA-H; ISPs can choose one based on their concerns. ***FIFA-S*** keeps the FIB size the smallest amongst the three, with very light FIB bursts and no FIB re-aggregation. ***FIFA-T*** is the fastest amongst the three, with relatively small number of FIB changes and fast re-aggregation. ***FIFA-H*** is a hybrid approach combining the advantages of both FIFA-S and FIFA-T. It has medium time cost compared to the other two schemes, and much lighter FIB burst than FIFA-T. Moreover, it does not perform any re-aggregations.

In the rest of this section, we describe our improved version of ORTC and the three FIFA algorithms.

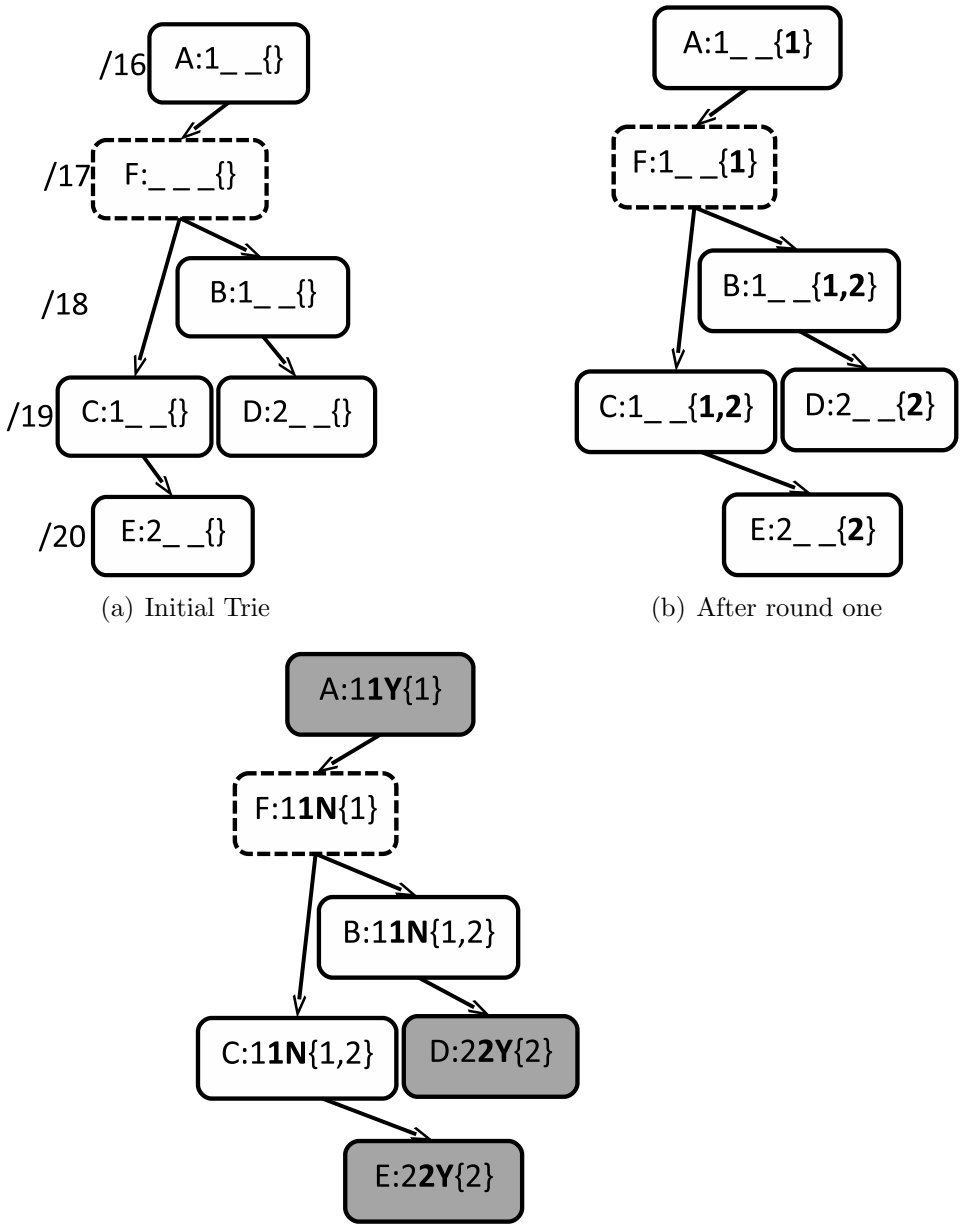
1.1 Improving ORTC Efficiency

FIB size can be aggregated optimally after applying the ORTC aggregation algorithm (The authors proved the optimality and strong forwarding correctness of

the algorithm in [29]). FIFA is based on the ORTC algorithm, but it addresses two inefficiencies in the latter: (1) the basic ORTC algorithm traverses the FIB tree three times, so it can be quite slow for a large FIB; and (2) ORTC uses a binary tree structure that could consume more memory than necessary when there are large gaps between address prefixes and the large number of tree nodes means slower tree traversals. We improved ORTC using two passes on a Patricia Trie [9]. A Patricia Trie is a space-optimized tree in which a child prefix can be longer than its parent prefix by more than one, thus eliminating unnecessary internal nodes. For example, Figure 3.2(a) shows the Patricia Trie representation of Table 2.1(a) – node C has a prefix length 19 while its parent F has a prefix length of 17. We tested both implementations using RouteView's data [18]. For the routing table of router 4.69.184.193 on 1/1/2011 (332,588 entries), our implementation is 2.5 times faster and uses only 44% of the memory consumed by the original implementation.

In order to distinguish the patricia trie-based ORTC algorithm from the basic ORTC, we use Round One (Figure 3.2(b)) and Round Two (Figure 3.2(c)) to represent its new passes. *Round One* is a depth-first traversal in post-order to merge next-hops (as in pass two) without normalizing the tree (otherwise we get a complete binary tree). *Round Two* is a depth-first traversal in pre-order to select next-hops (as in pass three) and it adds new tree nodes to maintain forwarding correctness.

In a Patricia Trie, we only use round one to obtain the same results as pass one and pass two of ORTC binary tree implementation. The task is challenging since it requires us to avoid expanding leaf nodes but be able to conduct the merge process correctly. Under this condition, we have to think of a way to simulate the merge process like running in a binary tree. Imagine in a complete binary tree after pass one, any internal node has two children and each of them has a merged next-hop set. The pass two recursively merges the two children sets into their parents until



(a) Initial Trie

(b) After round one

(c) After round two

Figure 3.2: Improved ORTC Aggregation Algorithm using Patricia Trie

reaching the tree root. Here we combine pass one and two together and emulate the same scenario. For Round One, in order to merge the next-hops correctly without expanding the trie, we compute a node's next-hop set by merging what would be the next-hop sets of its *imaginary* children if there is a complete binary tree. Let $S(n)$ be the next-hop set of node n, $S_l(n)$ and $S_r(n)$ be the next-hop set of n's imaginary left and right child, respectively. Then $S(n) = merge(S_l(n), S_r(n))$. In Figure 3.2(b), $S(n)$ is the last value associated with each node.

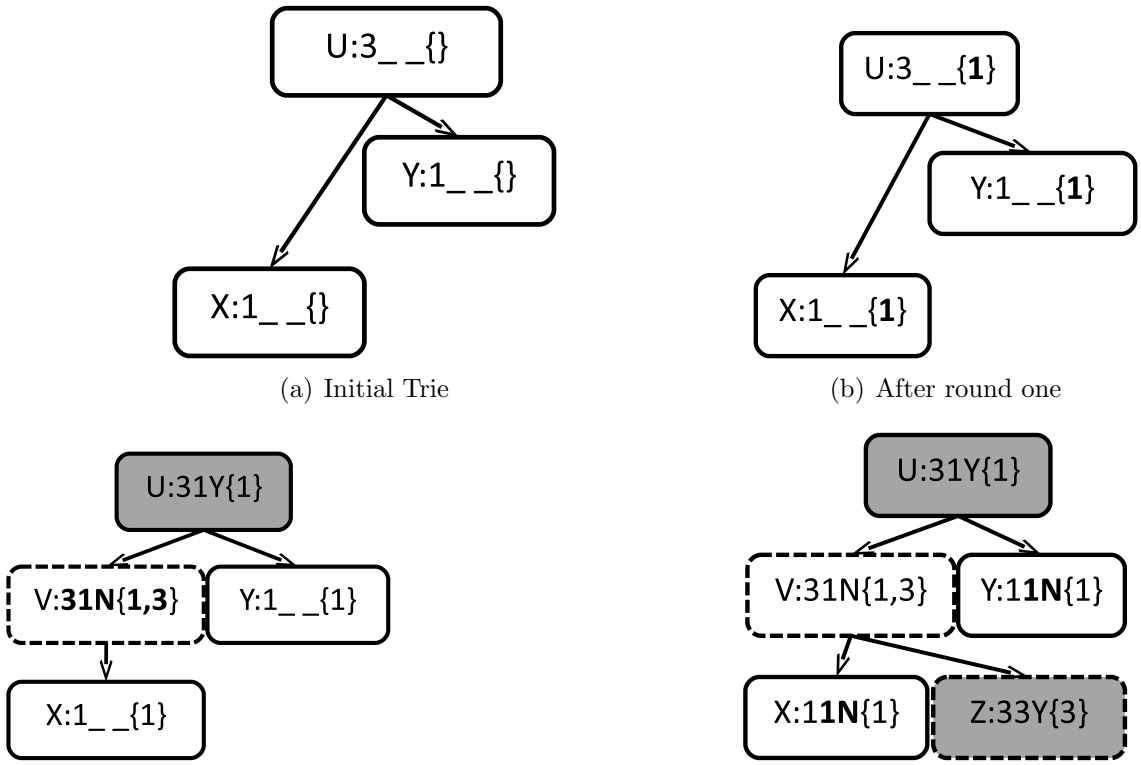
Below we explain how to compute $S_l(n)$. Let $H(n)$ be the original next-hop of n, and d be the difference between the prefix length of a node and that of its *actual* left child. There are four possible cases: no left child, $d = 1$, $d = 2$, and $d > 2$. In each case, the calculation follows a simple rule explained below (all the examples refer to the FIB tree in Figure 3.2(b)). The S_l value is assigned based on the following rules. The rules can be proven by expanding the part of a trie that includes the parent and the child into a complete binary structure and applying the merging rules to it.

1. No left child: S_l is derived from the original next-hop of the parent node, since the child was to be created from tree normalization. For example, C has no left child, so $S_l(C) = \{H(C)\} = \{1\}$.

2. $d = 1$: S_l is the next-hop set of the actual left child. For example, $d = 1$ for A and F, so $S_l(A) = S(F) = \{1\}$.

3. $d = 2$: S_l is the merged next-hops of the parent's original next-hop and the actual left child's next-hop set. For example, $d = 2$ between F and C, so $S_l(F) = \{H(F)\} \cap S(C) = \{1\} \cap \{1, 2\} = \{1\}$. Figure 2.2(c) depicts it clearly in a binary tree.

4. $d > 2$: S_l is a set containing only the original next-hop of the parent node. If the length difference is greater than two, then the dominated next-hop is the parent original next-hop.



(a) Initial Trie (b) After round one

(c) Create V in round two (d) Create Z in round two

Figure 3.3: Create New Nodes Under Certain Conditions

We then obtain $S_r(n)$ using the same procedure, and calculate $S(n)$ by merging $S_l(n)$ and $S_r(n)$. From bottom leaf nodes, we apply the merge rules all the way to root node of the trie, and will obtain identical merged next-hop sets as in a complete binary tree(except those unnecessary internal nodes) as shown in Figure 3.2(b) and Figure 2.2(c). For example, since $d = 1$ between F and B, $S_r(F) = S(B) = \{1, 2\}$. Therefore, $S(F) = S_l(F) \cap S_r(F) = \{1\} \cap \{1, 2\} = \{1\}$.

Round Two goes through similar steps as pass three to select the next-hop of each node. In addition, it creates a new node when $H(n) \neq H'(n)$, where $H(n)$ and $H'(n)$ are the original and selected next-hop of node n, and one of the following two conditions is satisfied:

1. $d \geq 2$: if n has a left (right) child with prefix length greater than n's length by at least 2, then a left (right) child under n is created. For example, in

Figure 3.3(b), $H(U) = 3 \neq H'(U) = 1$ and $d = 2$ between node U and X, then we need to create a new node V as a left child of U.

2. One child is missing: if n has no left (or right) child, then a left (or right) child under n is created. For example, in Figure 3.3(c), $H(V) = 3 \neq H'(V) = 1$ and V has no right child, so we create a new node Z as a right child of V, as shown in Figure 3.3(d). So after this step, only prefixes on node U with next-hop 1 and Z with next-hop 3 will be placed into FIB.

Otherwise, we do not need to create new nodes. After the two rounds, we obtain the same set of aggregated FIB entries as the original ORTC does with much fewer nodes in general. For example, in Figure 3.2(c), we did not create any new node because $H(n) = H'(n)$ for all the nodes, and only six nodes are created compared to 11 nodes in Figure 2.2(d). But the Trie implementation in Figure 3.2(c) saves a lot of space through not using unnecessary internal nodes.

1.2 FIFA-S

FIFA-S keeps the aggregated FIB size optimal after every update. A naive way to do so is to perform the ORTC aggregation on the entire FIB trie upon every update, but this would be too time-consuming. A better approach is to update only those parts of the FIB trie that may have been impacted by the update. We follow this approach in both the optimal size update handling algorithm (BasicOptSize) we proposed in 2010 [53] and FIFA-S, but FIFA-S is eight times faster than BasicOptSize (see Section 2) and its heaviest FIB burst (i.e., number of FIB changes caused by a single route change) is only 1/10 of that in BasicOptSize. Below we first describe how BasicOptSize works and then show FIFA-S' improvements.

The BasicOptSize algorithm goes through the following steps, after applying the update to the corresponding node:

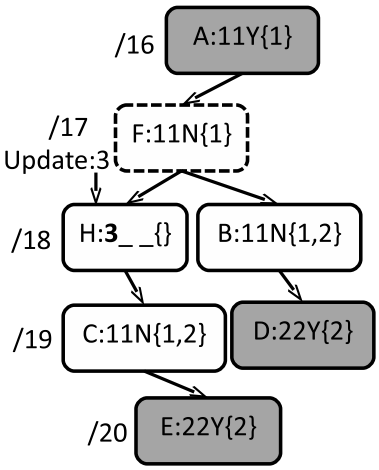


Figure 3.4: Updating node H upon receiving a new route

Table 3.1: Unaggregated and aggregated FIB entries after an update

(a) Original FIB Entries

Label	Prefix	Next-hop
A	141.225.0.0/16	1
B	141.225.64.0/18	1
C	141.225.32.0/19	1
D	141.225.96.0/19	2
E	141.225.48.0/20	2
H	141.225.0.0/18	3

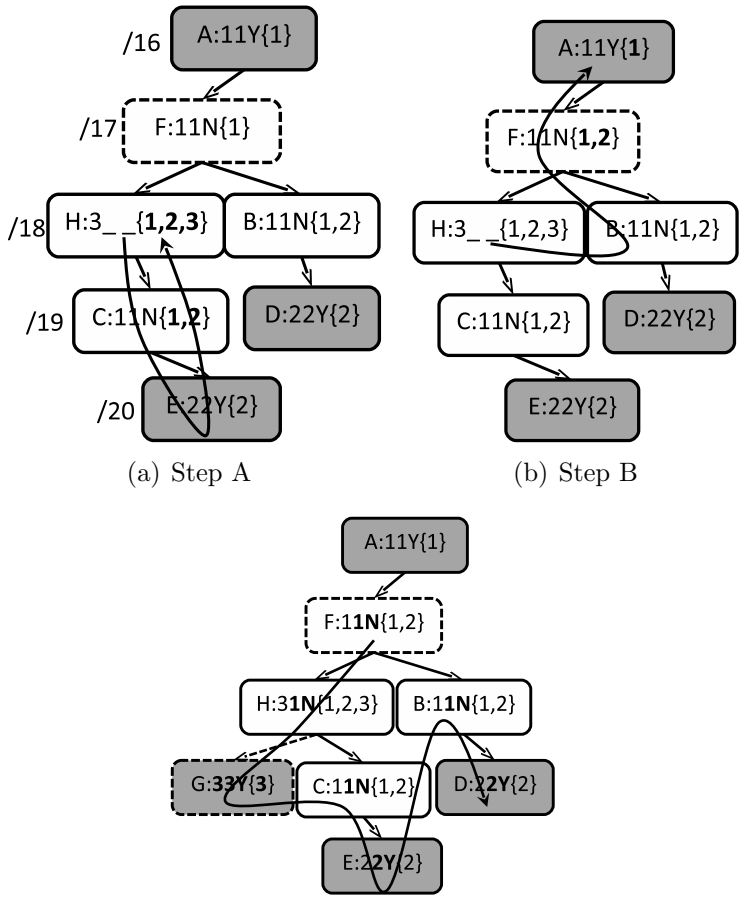
(b) Aggregated FIB Entries

Label	Prefix	Next-hop
A	141.225.0.0/16	1
D	141.225.96.0/19	2
E	141.225.48.0/20	2
G	141.225.0.0/19	3

1. Step A: on the subtree rooted at the *updated node*, merge the next-hops using a depth-first traversal in post-order. This is basically a *Round One* operation on a subtree;

2. Step B: for each ancestor node *above* the updated node, merge its next-hops until the node's new next-hop set is the same as its old next-hop set. We call this node the "highest changed node";

3. Step C: on the subtree rooted at the *highest changed node*, select the next-hops using a depth-first traversal in pre-order. This is a *Round Two* operation on a subtree.



(a) Step A (b) Step B

(c) Step C

Figure 3.5: Steps in BasicOptSize

To illustrate BasicOptSize, we add a new entry to our example FIB (H in Table 3.1(a)). Figure 3.4 shows the FIB trie after updating node H (its type is changed to REAL and its next-hop from 1 to 3). Figure 3.5 shows Steps A, B and C. After Step C, the aggregated FIB contains three of the five original entries, A, D, and E, and a new entry G (Table 3.1(b)).

Figure 3.5 shows that Step B and C need to update the entire subtree rooted at H and F, respectively. To reduce the number of nodes visited on these subtrees, FIFA-S takes advantage of the following two properties (see the appendice for their proofs):

***Property 1**: The result of Step A will be the same without updating any subtrees rooted at REAL nodes.*

Property 2*: The result of Step C will be the same without updating any subtree with these properties: (1) the next-hop sets did not change in any nodes on the subtree in Step A; and (2) the selected next-hop of the subtree root did not change.*

Moreover, FIFA-S adopts the following rules and it has considerably fewer FIB changes than BasicOptSize (Section 2).

***Property 3**: In Step C, selecting a node's next-hop from its next-hop set using the following prioritized rules can reduce the number of FIB changes: (a) the next-hop selected by the nearest ancestor with IN_FIB status (this is for FIB size optimization); (b) the old selected next-hop; (c) the original next-hop; and (d) if none of those are found in the next-hop set, sort the set and pick the first one instead of random selection.*

We call the improved procedures *Step A', B' and C'*. Figures 3.5 and 3.6 show that (1) BasicOptSize and FIFA-S have the same aggregation results; (2) E was skipped in Step A'; and (3) D and E were skipped in Step C'.

We present the pseudo code of FIFA-S in Procedures 1 - 6. Function *mergeNexthopsBelowNode* is Step A', *mergeNexthopsAboveNode* is Step B' and *selectNexthop* is Step C'. Note that we associate a flag *optimal* with each node to indicate whether Step C' is needed in the subtree of this node.

In the next two sections, we show how Properties 1-3 can be used in FIFA-T and FIFA-H to reduce computation overhead and keep the aggregated trie stable.

1.3 FIFA-T

FIFA-T aims to shorten the FIB update time by localizing the changes on the FIB trie while maintaining strong forwarding correctness. The trade-off is that the FIB size will not be optimal. As more updates come, the FIB size will increase until it

Procedure 1 *main*(*type*) function

```
Build initial FIB trie based on unaggregated FIB
Run improved ORTC on the FIB trie to obtain aggregated FIB
for each update do
    if Announcement then
        Lookup the corresponding node and create it if non-existent
        Update the next-hop of the current node
        node.type ← REAL
    else
        Lookup the corresponding node and return if non-existent
        Remove the next-hop of the current node
        node.type ← AUXILIARY
    if (type = ALG_FIFA_T) ∨ (type = ALG_FIFA_H) then
        node.optimal ← 0 for all ancestors of the current node
    switch (type)
    case ALG_FIFA_S:
        FIFA_S(node)
    case ALG_FIFA_T:
        FIFA_T(node)
    case ALG_FIFA_H:
        FIFA_H(node)
    end switch
```

Procedure 2 $FIFA\_S(node)$ function

```
realAncestor ← nearestRealAncestor(node)
mergeNexthopsBelowNode(node, realAncestor)
highestNode ← mergeNexthopsAboveNode(node, ALG_FIFA_S)
infibAncestor ← nearestINFIBAncestor(highestNode)
selectNexthop(highestNode, infibAncestor)
```

Procedure 3 $mergeNexthopsBelowNode(node, realAncestor)$

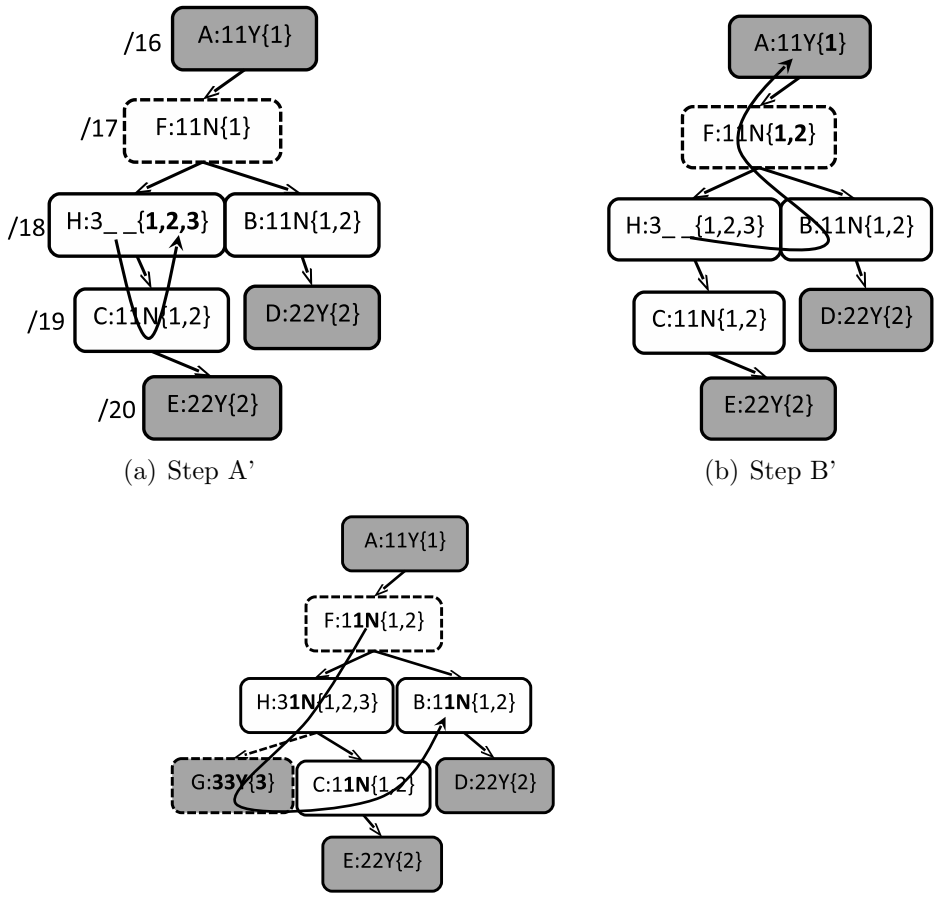
```
node.optimal ← 0
l ← node.l
r ← node.r
if (l ≠ NULL) ∧ (l.type ≠ REAL) then
    mergeNexthopsBelowNode(l, realAncestor)
if (r ≠ NULL) ∧ (r.type ≠ REAL) then
    mergeNexthopsBelowNode(r, realAncestor)
if (node.type ≠ REAL) then
    node.originalNexthop ← realAncestor.originalNexthop
node.mergedNexthops ← merge(l, r)
```

Procedure 4 $mergeNexthopsAboveNode(node, type)$

```
parent ← node.parent
while parent do
    if (type = ALG_FIFA_H) ∧ (parent.length ≤ CAP) then
        Return node
    old ← parent.mergedNexthops
    new ← merge(parent.l, parent.r)
    if old = new then
        Return node
    node ← parent
    parent ← node.parent
    if type = S then
        node.optimal ← 0
Return node
```

Procedure 5 $selectNexthop(node, ancestor)$

```
oldStaus ← node.status
oldNexthop ← node.selectedNexthop
if ancestor.selectedNexthop ∈ node.mergedNexthops then
    node.selectedNexthop ← ancestor.selectedNexthop
    node.staus ← NON_FIB
else
    if oldNexthop ∈ node.mergedNexthops then
        node.selectedNexthop ← oldNexthop
    else if node.originalNexthop ∈ node.mergedNexthops then
        node.selectedNexthop ← node.originalNexthop
    else
        node.selectedNexthop ← node.mergedNexthops[0]
    node.staus ← IN_FIB
updateFIB(oldStatus, oldNexthop, node)
if (oldNexthop = node.selectedNexthop) ∧ (node.optimal = 1) then
    Return
if node.status = IN_FIB then
    ancestor ← node
if (node.l = NULL) ∧ (node.r = NULL) then
    Return
if node.selectedNexthop ≠ node.originalNexthop then
    generateNewNode(node)
if node.l ≠ NULL then
    selectNexthop(node.l, ancestor)
if node.r ≠ NULL then
    selectNexthop(node.r, ancestor)
node.optimal ← 1
```



(a) Step A'

(b) Step B'

(c) Step C'

Figure 3.6: Steps in FIFA-S

reaches a threshold, *e.g.*, 90% of the FIB memory in the line card. At this point, a re-aggregation is performed on the FIB trie from the root to optimize the FIB size. On the surface, FIFA-T is very similar to the minimal time update handling algorithm (BasicMinTime) we proposed [53]. However, there are two important differences that make FIFA-T more efficient: (a) FIFA-T utilizes the three properties described in Section 1.2; and (b) FIFA-T's re-aggregation is performed on the aggregated FIB trie, but BasicMinTime has to destroy the old aggregated FIB trie, and build a new one from the unaggregated FIB. Our results show that it uses 40% less time than BasicMinTime and generates only 1.1 FIB changes per routing

Procedure 6 $updateFIB(oldStatus, oldNexthop, node)$

```
1: if oldStatus ≠ node.status then
2:     if node.status = NON_FIB then
3:         Delete the prefix and next-hop from FIB
4:     else
5:         Add the prefix and next-hop to FIB
6: else if oldNexthop ≠ node.selectedNexthop then
7:     if node.status = IN_FIB then
8:         Update the corresponding next-hop to FIB
```

update.

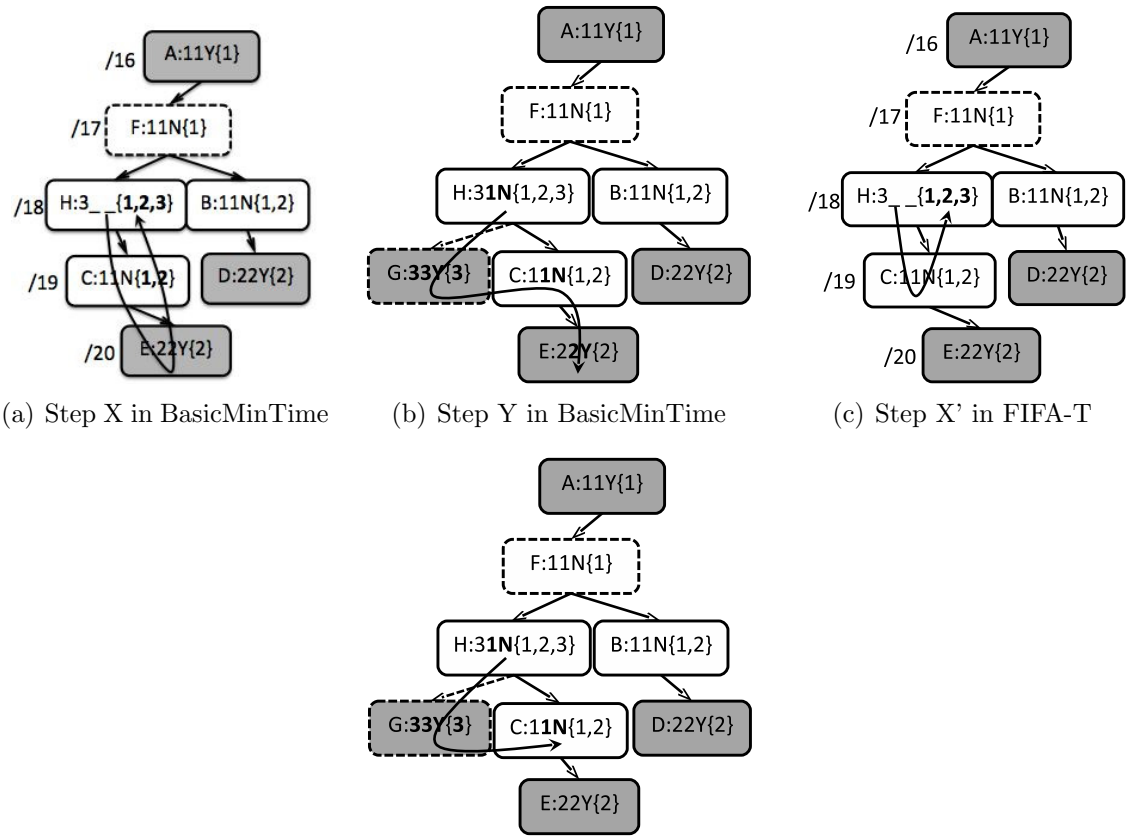
FIFA-T works as follows: (1) before the threshold is reached, perform the following (the less efficient procedures in BasicMinTime are called *Step X* and *Y*) –

- Step X': on the subtree rooted at the *updated node*, merge the next-hops using a depth-first traversal in post-order, skipping REAL nodes and their subtrees (based on Property 1);
- Step Y': on the subtree rooted at the *updated node*, select the next-hops (following rules based on Property 3) using a depth-first traversal in pre-order, skipping REAL nodes with optimal flag set to 1 as well as their subtrees (based on Property 2).

(2) when the threshold is reached, re-aggregate the trie from its root incorporating the three properties to obtain an optimal trie. The pseudo code is in Procedures 1, 7, and 3 - 6.

Procedure 7 $FIFA\_T(node)$ function

```
1: if Threshold then
2:     Do re-aggregation on the FIB trie from the root
3: else
4:     realAncestor ← nearestRealAncestor(node)
5:     mergeNexthopsBelowNode(node, realAncestor)
6:     infibAncestor ← nearestINFIBAncestor(node)
7:     selectNexthop(node, infibAncestor)
```



(a) Step X in BasicMinTime (b) Step Y in BasicMinTime (c) Step X' in FIFA-T

(d) Step Y' in FIFA-T

Figure 3.7: Steps in BasicMinTime and FIFA-T. Step X and Y are steps for BasicMinTime, while Step X' and Y' are steps for FIFA-T.

Figure 3.7 illustrates the differences between BasicMinTime and FIFA-T, e.g., node E was skipped in Step X' and Y'.

1.4 FIFA-H

In addition to FIFA-S and FIFA-T, we propose FIFA-H, a hybrid scheme that achieve a good balance among aggregation speed, FIB size and number of FIB changes. In this approach, a FIB size threshold and a CAP are set at the beginning. For each update, FIFA-H performs three steps - U, V, W (or W') as follows (Figure 3.8):

- Step U: merge the next-hops below the updated node (same as Step A' in FIFA-S and Step X' in FIFA-T);

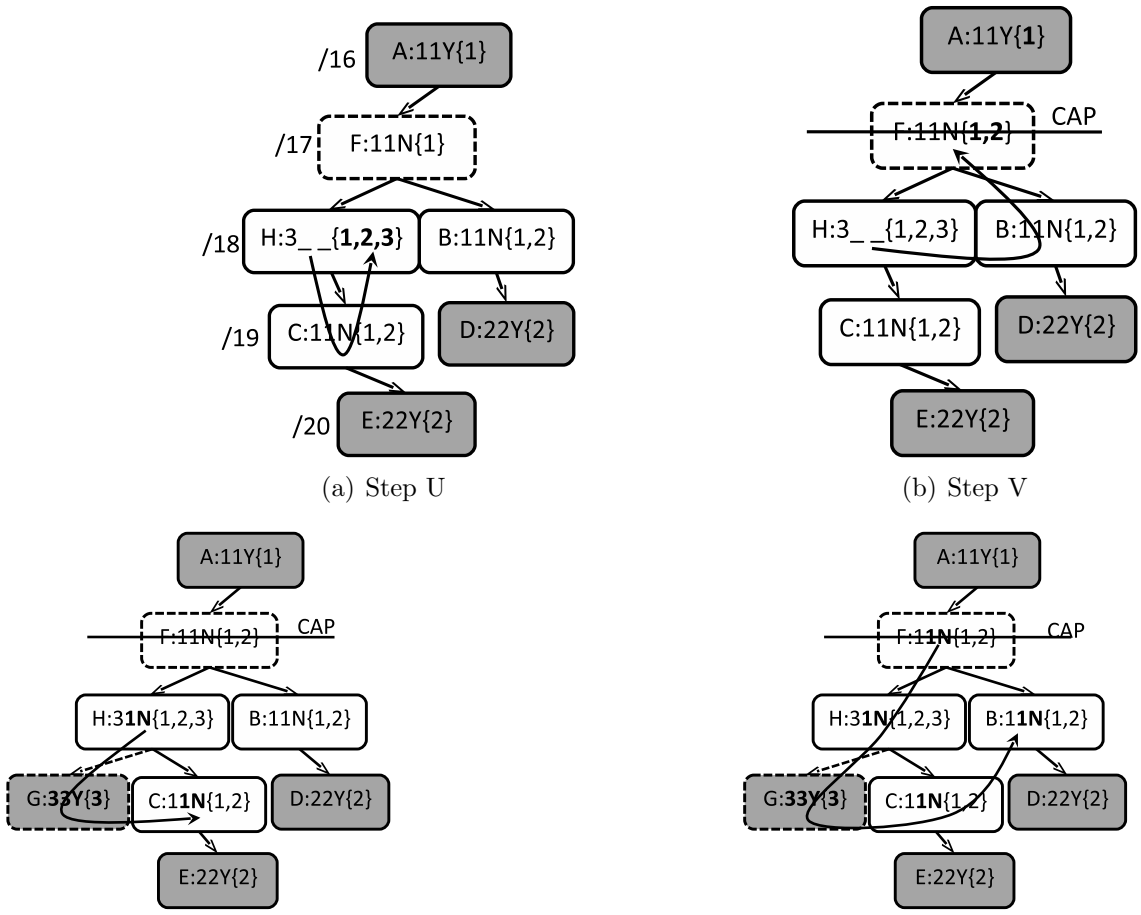
- Step V: merge the next-hops above the updated node up to the highest changed node whose prefix length is less than or equal to CAP, called the CAP node, which limits the computation overhead and the number of FIB changes compared to FIFA-S;

- Step W or W': if the threshold is not reached, this step (W) performs next-hop selection on the subtree rooted at the current updated node (*SaveTime* mode). Otherwise, this step (W') will start from the CAP node for next-hop selection (*ReduceSize* mode).

FIFA-H incurs less computation time and fewer FIB changes than FIFA-S, and has smaller FIB bursts than FIFA-T (Section 2). It has no lengthy re-aggregations, thus avoiding potential problems during re-aggregation, e.g., packet losses.

Procedure 8 $FIFA\_H(node)$ function

1: $realAncestor \leftarrow nearestRealAncestor(node)$
2: $mergeNexthopsBelowNode(node, realAncestor)$
3: $capNode \leftarrow mergeNexthopsAboveNode(node, ALG\_FIFA\_H)$
4: **if** ThresholdReached **then**
5: $\quad node \leftarrow capNode$
6: $infibAncestor \leftarrow nearestINFIBAncestor(node)$
7: $selectNexthop(node, infibAncestor)$



(a) Step U (b) Step V

(c) Step W (d) Step W'

Figure 3.8: FIFA-H: the first two steps are Step U and V. The third step is Step W before reaching the threshold and Step W' after reaching the threshold.

2 Evaluation

In this section, we evaluate the performance improvement of FIFA over BasicOptSize, BasicMinTime, as well as SMALTA [79], another ORTC-based FIB aggregation scheme. We also compare the three FIFA algorithms so that users can choose the right algorithm based on their own needs. We verified the correctness of our results by checking that every address has the correct next-hop after aggregation.

2.1 Methodology

We used RIBs and routing updates from 01/01/2011 to 12/31/2011 in the Routeviews [18] route-views2 data archive. Since the routing updates do not contain next-hop IP address information, we use next-hop ASes to approximate next-hop routers (as in [82, 53]) and we have used internal routing information from a tier-1 ISP to verify that our approach closely approximates the results using IGP next-hops. In order to show the worst-case performance, we present the results from 4.69.184.193, a router in the tier-1 ISP *Level 3*, because this router has the most AS neighbors (2876 and 3151 on 01/01/2011 and 12/31/2011, respectively) among all 36 routers. In general, more neighbors mean more next-hops the prefixes can have, which may lead to lower FIB aggregation performance. We also tested another router, 216.218.252.164, from Hurricane Electric with 1549 AS neighbors, and obtained similar results. In practice, a router has tens or at most hundreds of interfaces.

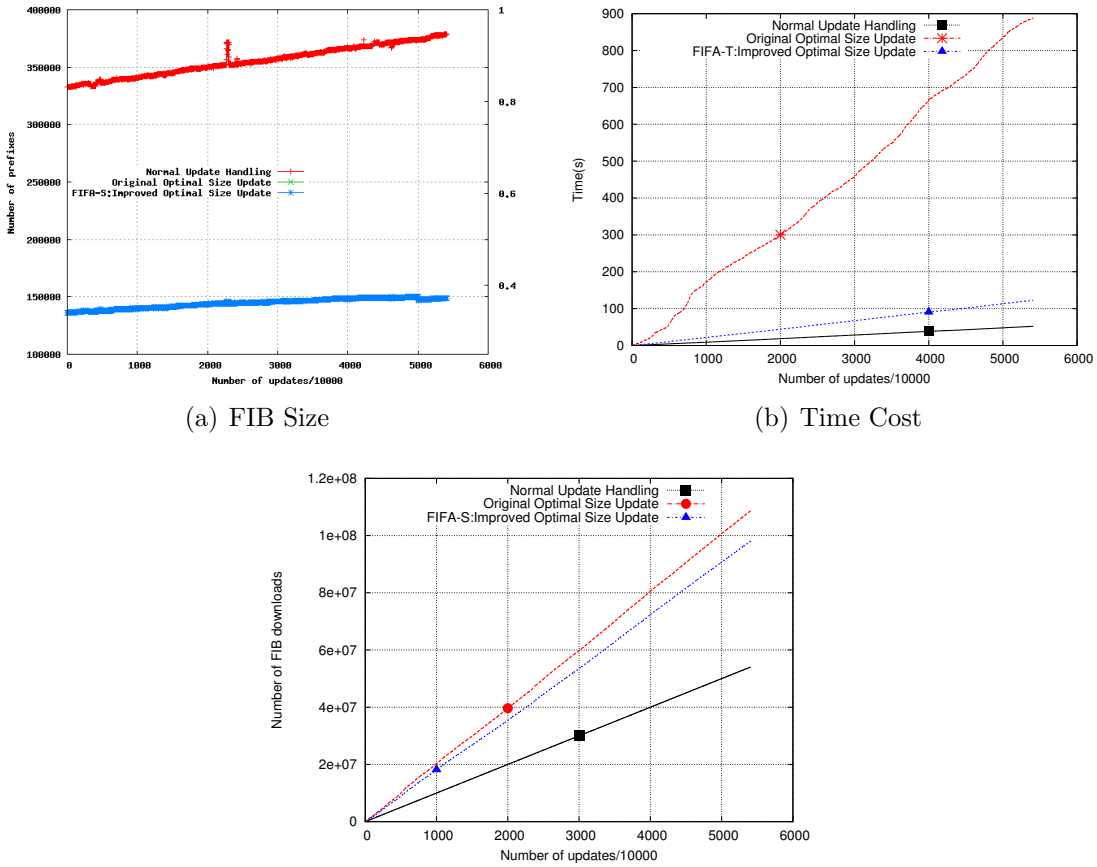
Before running the update handling, we filtered out all duplicate update messages because in practice the duplicate message will be filtered out and prevented from entering main RIB and FIB updates. Before the filtering, there were 155,425,645 updates, and the number was reduced to 54,095,965 after the preprocessing. Thus any update handling algorithm which is comparable to ours should filter out the duplicate messages first. Also we use Next AS Hop as the next-hop to evaluate the performance of update handling.

In the results, the forwarding correctness was guaranteed by our own implemented forwarding equivalence verification algorithm, in which we used an efficient method to go through the whole 32-bit IPv4 space to do the comparisons between original FIB and aggregated FIB. We use the following four performance metrics: (1) FIB Size: total number of entries in FIB; (2) Time Cost: time to apply

routing changes to the FIB including re-aggregation time, if any; (3) FIB Changes: total number of FIB updates caused by all routing updates; and (4) FIB burst: number of FIB changes caused by one route change. The evaluation was done on a machine with an Intel Core 2 Quad 2.83GHz CPU.

2.2 Summary of Findings

First, notice that in year 2001, there were about 54 million updates, and on average 1.7 updates per second, however, FIFA algorithms could handle 500 updates (2 μs/update) in the worse case, which indicates that our algorithms are far good enough to handle normal updates, even FIB bursts with hundreds or thousands of updates. Second, FIFA algorithms have significantly enhanced the performance over existing ones. FIFA-S improves the time efficiency of BasicOptSize by 8.22 times and keeps the FIB size optimal. It is mostly useful when the FIB memory size is close to its optimal aggregated size, when FIFA-T will trigger too many re-aggregations. FIFA-T is the fastest among the three schemes; it is suitable when the FIB memory is much larger than the optimal aggregated size. FIFA-H is a well-balanced scheme with medium running time and FIB burst size. Compared with SMALTA, ***all FIFA algorithms are faster and have smaller FIB bursts***. In addition, ***FIFA-T and FIFA-H incur smaller total number of FIB changes than SMALTA***. Technically, there are mainly three factors leading to the fast FIB aggregation in FIFA. First of all, SMALTA requires rebuilding the FIB aggregation tree from scratch during re-aggregation, which is time consuming. In FIFA-T, the re-aggregations are very fast, because it does not require rebuilding the FIB aggregation. In FIFA-S and FIFA-H, there is no re-aggregation. Secondly, SMALTA uses a binary tree data structure, but we use a patricia trie. The binary tree has many more nodes to traverse than the patricia trie, thus incurring more computation overhead. Thirdly, FIFA has the two important features which are not



(a) FIB Size

(b) Time Cost

(c) FIB Changes

Figure 3.9: FIFA-S vs. BasicOptSize (*Normal* refers to no aggregation)

applicable to SMALTA and they can further help reduce more redundant node accesses but keep the same aggregated size.

2.3 FIFA-S vs. BasicOptSize

We first compare FIFA-S with BasicOptSize. Figure 3.9(a) shows the FIB size. Since both schemes achieve optimal FIB size, their lines overlap with each other ending below 150,000. The top line shows the unaggregated FIB size, which increased from 332,588 to 378,728 during the year. In other words, ***either scheme reduced the FIB size by about 60%***. If the unaggregated FIB size increases at

Table 3.2: FIB Burst Distribution Comparison between FIFA-S and BasicOptSize

Burst Size	Min	Max	Median	=0	≤1	≤10	All
BasicOptSize	0	6,226	1	6,914,934 (12.78%)	38,185,015 (70.58%)	52,795,683 (97,59%)	54,095,965 (100%)
FIFA-S	0	568	1	6,961,449 (12.87%)	38,645,578 (71.43%)	53,318,607 (98.56%)	54,095,965 (100%)

the current rate (about 13.9%), it will take 7.5 more years for the aggregated FIB size to reach the current unaggregated FIB size (as of 12/31/2011). As aforementioned, this is a conservative and underestimated ratio since we were using around 3,000 distinct next AS hops as outgoing interfaces. In real practices, it may be compressed much more than this peer.

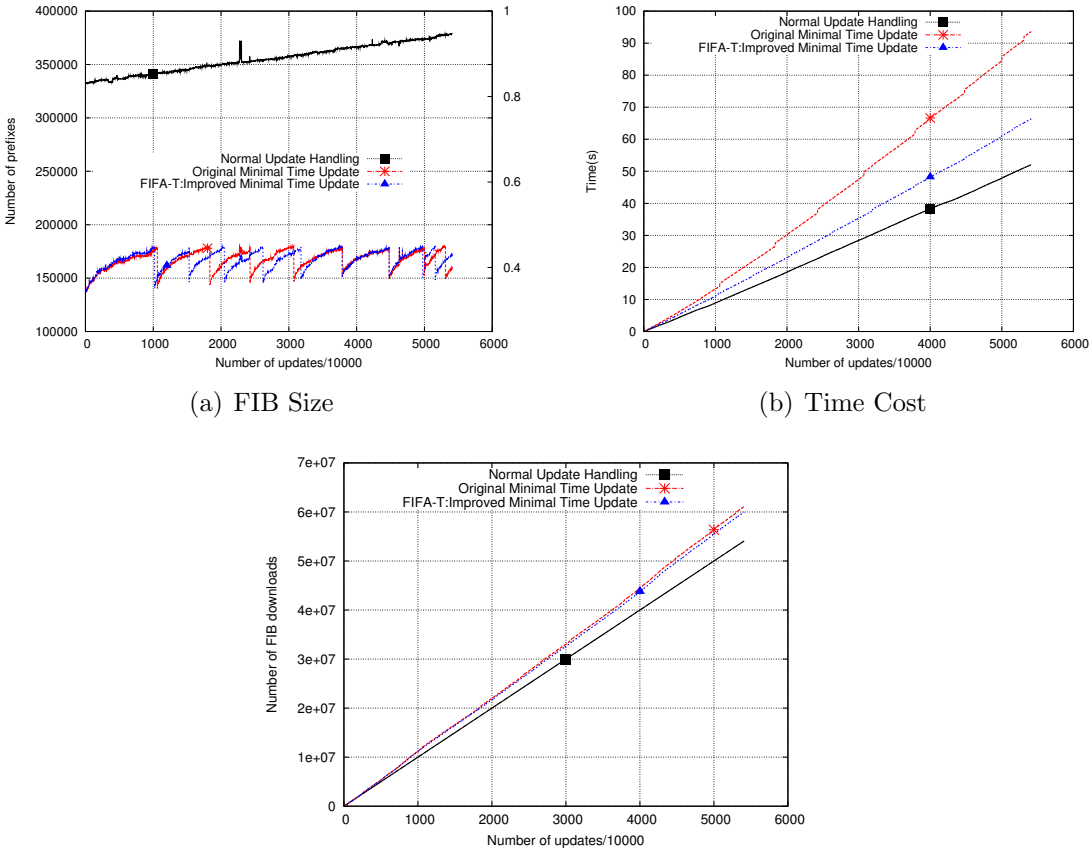
Figure 3.9(b) shows that ***FIFA-S is more than 8.22 times faster*** (108s in total or 2μs/update) ***than BasicOptSize*** (888s in total or 16.4μs/update). The time cost of FIFA-S is very close to the bottom line, which corresponds to the time cost to update an unaggregated FIB. This suggests that it is feasible to deploy FIFA-S in an operational router.

Figure 3.9(c) shows that the total number of FIB changes in FIFA-S is only about 1.8 times of that in an unaggregated FIB, and this ratio is very stable.

Table 3.2 shows the FIB burst distribution. In both schemes, about 98% of the FIB bursts have no more than 10 FIB changes. Moreover, ***FIFA-S' largest FIB burst (568) is less than 10% of that in BasicOptSize (6,226)***.

2.4 FIFA-T vs. BasicMinTime

In Figure 3.10 and Table 3.3, we compare FIFA-T with BasicMinTime and observe the following: (a) their FIB size oscillates between the optimal size and the configured threshold, and FIFA-T triggers only 9 fast re-aggregations during the entire year (Figure 3.10(a)); (b) ***FIFA-T uses 40% less time than***



(a) FIB Size

(b) Time Cost

(c) FIB Changes

Figure 3.10: FIFA-T vs. BasicMinTime (*Normal* refers to no aggregation)

BasicMinTime (Figure 3.10(b)); and (*c*) FIFA-T's largest FIB burst is much smaller than that in BasicMinTime (Table 3.3).

2.5 Comparison among FIFA Algorithms and with SMALTA

Below we compare FIFA algorithms and SMALTA.

a). FIB Size: Figure 3.11(a) shows that (1) FIFA-S has the smallest FIB size; (2) FIFA-T and SMALTA oscillate between the optimal size and the threshold, and (3) FIFA-H tends to stay around the threshold with no re-aggregation.

b). Time Cost: Figure 3.11(b) shows that (1) SMALTA takes the most time – 237.23s, which includes 11 full tree re-aggregations (158.62s or 14.3s per

Table 3.3: FIB Burst Distribution Comparison between FIFA-T and BasicMinTime

Burst Size	Min	Max	Median	=0	≤1	≤10	All
BasicMinTime	0	149,815	1	6,080,983 (11.24%)	49,611,562 (91.71%)	53,913,320 (99.66%)	54,095,973 (100%)
FIFA-T	0	69,526	1	6,150,664 (11.36%)	49,704,177 (91.88%)	53,919,736 (99.67%)	54,095,965 (100%)

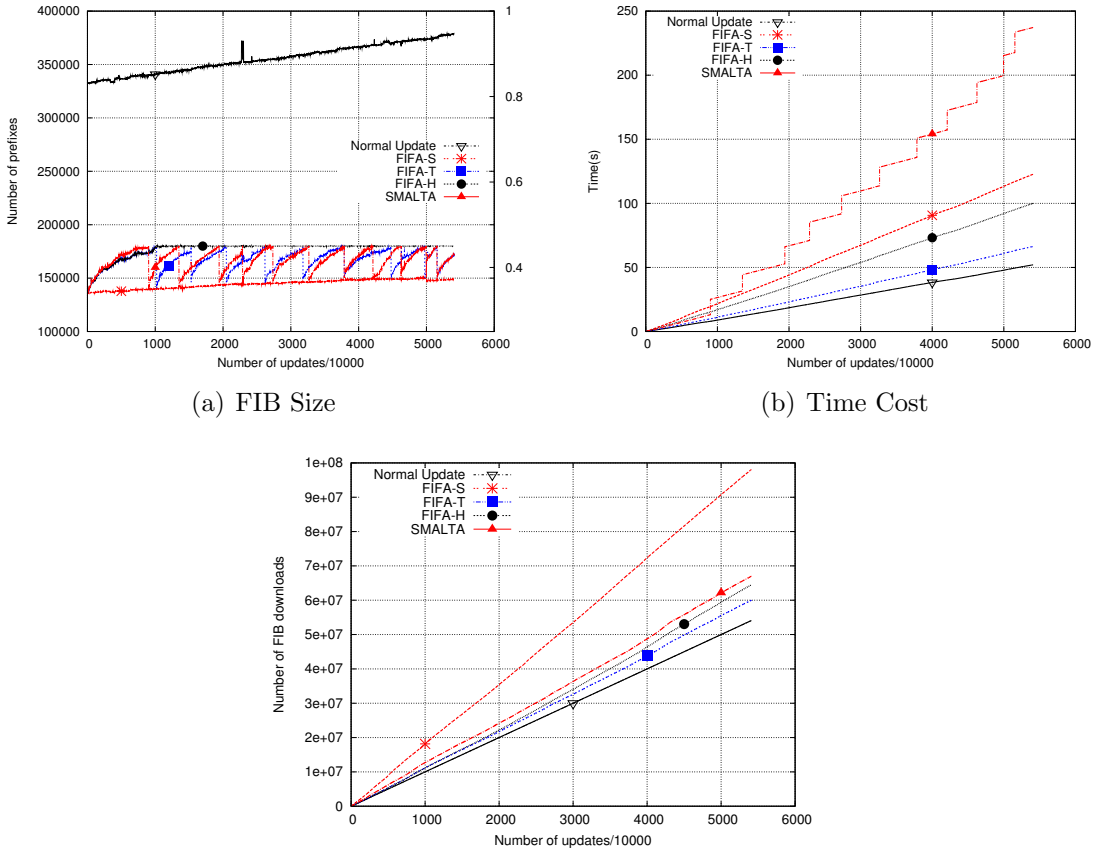
Table 3.4: FIB Burst Distribution Comparison among three FIFA schemes and SMALTA

Burst Size	Min	Max	Median	=0	≤1	≤10	All
FIFA-S	0	568	1	6,961,449 (12.87%)	38,645,578 (71.43%)	53,318,607 (98.56%)	54,095,965 (100%)
FIFA-T	0	69,526	1	6,150,664 (11.36%)	49,704,177 (91.88%)	53,919,736 (99.67%)	54,095,965 (100%)
FIFA-H	0	1,182	1	6,232,328 (11.52%)	48,997,278 (90.57%)	53,784,161 (99.42%)	54,095,965 (100%)
SMALTA	0	72,856	1	4,456,410 (8.23%)	48,297,973 (89.28%)	53,873,603 (99.58%)	54,095,976 (100%)

re-aggregation); (2) FIFA-T is the fastest – 66s, which includes 9 fast tree re-aggregations with 0.2s for each (FIFA-T is 70 times faster than SMALTA in re-aggregation efficiency); and (3) FIFA-S (108s) and FIFA-H (100s) have similar time cost.

c). FIB Changes: Figure 3.11(c) shows that (1) FIFA-T and FIFA-S have the lowest and highest total number of FIB changes, respectively; (2) SMALTA has a slightly higher number of FIB changes than FIFA-H.

d). FIB Bursts: Table 3.4 shows that (1) most route changes cause zero or one FIB change, and about 99% of FIB bursts have less than 10 FIB changes; (2) FIFA-T usually has small FIB bursts, but they can get very large (69,526); (3) with FIFA-S and FIFA-H, the FIB bursts have at most 568 and 1,182 FIB changes, respectively; and (4) SMALTA has the largest FIB burst (72,856).



(a) FIB Size

(b) Time Cost

(c) FIB Changes

Figure 3.11: FIFA Algorithms vs. SMALTA (*Normal* refers to no aggregation)

2.6 Comparisons with Different Thresholds

Table 3.5 shows different performance results for different thresholds for all schemes and obtains similar results as above. We make the following observations: (a) The BasicMinTime update scheme is very sensitive to threshold change for its running time. (b) The BasicMinTime update scheme always has more FIB changes than FIFA-T for different thresholds. (c) The BasicMinTime update scheme always has longer running time than FIFA-T for different thresholds. (d) FIFA-T has slightly higher number of fast re-aggregations than the number of slow re-aggregations in the BasicMinTime update scheme. (e) The running time of FIFA-H is more

Table 3.5: Performance comparisons for different thresholds

Scheme	FIB Size	Time Cost (s)	FIB Downloads	Heaviest FIB Burst	Re-aggregation times
Normal Update	378,728	52.04	54,090,000	1	0
BasicOptSize Update	148,998	888.20	108,692,712	6,226	0
FIFA-S	148,998	108.49	98,088,041	568	0
BasicMinTime Update	160,000	135.37	67,173,411	149,834	51 slow
FIFA-T	160,000	76.70	60,673,484	40,234	57 fast
FIFA-H	160,000	105.13	77,823,595	667	0
SMALTA	160,000	981.52	67,826,947	51,329	62 very slow
BasicMinTime Update	180,000	93.60	61,112,373	149,815	8 slow
FIFA-T	180,000	66.34	59,999,817	69,526	9 fast
FIFA-H	180,000	100.00	64,504,256	1,182	0
SMALTA	180,000	237.23	67,003,405	76,941	11 very slow
BasicMinTime Update	200,000	83.86	60,337,631	149,341	2 slow
FIFA-T	200,000	64.84	59,863,910	97,959	3 fast
FIFA-H	200,000	97.10	60,735,995	1,768	0
SMALTA	200,000	138.36	66,697,953	111,986	4 very slow

sensitive to threshold change than FIFA-T. (f) FIFA-H has longer running time, higher total FIB changes, and smaller FIB burst than FIFA-T.

Also we tested different thresholds to compare the performance between FIFA and SMALTA. As expected, FIFA schemes outperforms SMALTA on all metrics with the exception that FIFA-S has more FIB changes because it always keeps the FIB size optimal for each update. However, FIFA-S always runs faster than SMALTA. For example, SMALTA needs to take 981.52s to finish all updates with 62 very slow full tree re-aggregations when the threshold is set as 160,000. However, our FIFA-T, FIFA-H and FIFA-S schemes only need to use 76.70s, 105.13s and 108.49s, respectively, to accomplish the same task. Notice that as the threshold is set lower, SMALTA takes much longer to do full tree re-aggregations, and its performance thus degrades sharply. On the contrary, FIFA schemes keep very stable running performance compared with the normal update handling.

2.7 FIB Lookup Performance and Memory Saving Efficiency

We use the software reference design of Tree Bit Map [30] to obtain the FIB memory access times and memory usage. The reference implementation uses an initial 8K entries with stride 13, and following stride 4 to create a 13, 4, 4, and 4 multi-bit trie. Each TBM node with stride 4 uses 8 bytes to store internal and external bit map, and an additional 32 bits to store the pointer to its children and results array. The lookup memory access times include trie lookup memory access and data access times. For lookup memory access times, each trie node access is considered as one memory access. The data access was postponed to the last step, and the value is one if there is a matching prefix in the trie(otherwise the data access time is 0).

We assume the traffic has a uniform distribution pattern. Figure 3.12 demonstrates the ratio of FIB size, FIB lookup memory access times, and real FIB memory usage after and before aggregation using 33 peers on 12/31/2011. All the RIB sizes for these peers are more than 360,000, with minimum FIB size 364,729, maximum 389,562, and median 381,188. For example, each point of the triangle line represents the value of FIB memory usage after aggregation over the one before aggregation. We make the following observations from the figure: (a)The aggregation can mainly reduce FIB memory usage, as well as FIB lookup memory access times. (b) The overall memory saving ratio of aggregation(triangle line, ranging from 21.29% to 52.24% with median 49.60%) is less than the FIB entry saving ratio (rectangle line, ranging from 13.28% to 39.57% with median 36.74%) ranging from 8.00% to 13.96% with median 12.62%, these results are very similar with SMALTA results [79]. (c) The overall FIB lookup memory access times (round circle line) after aggregation do not improve much, and are slightly less than that before aggregation. Compared with SMALTA results, ours are very different, because SMALTA does not count the data (results array) memory access times as

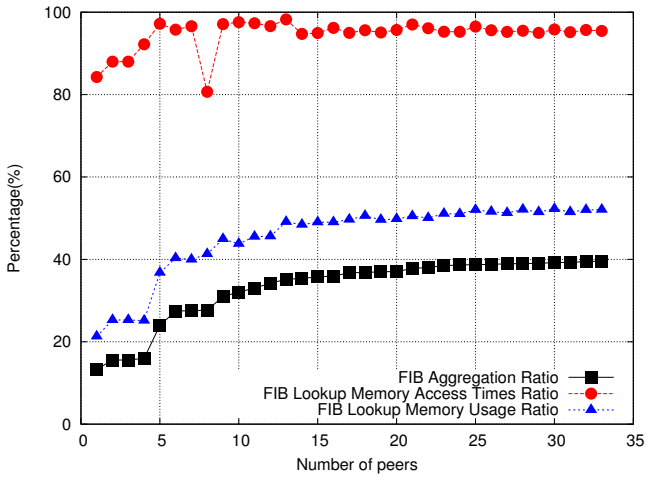


Figure 3.12: FIB Lookup Performance Improvement

well as the memory access times of non-matching prefixes in the TreeBitMap trie, as confirmed with SMALTA authors.

3 Forwarding Correctness Verification

In order to guarantee the forwarding correctness for aggregation and update handling, we implemented an efficient verification algorithm, in which the whole IPv4 space will be checked with prefixes rather than individual IP addresses. Through different settings, we can verify both strong and weak forwarding correctness for any two routing tables. Given any two router tables with prefixes and corresponding next-hops, the verification algorithm can not only tell if the two routing tables have forwarding equivalence, but also output the forwarding difference if they have different forwarding behaviors.

We noticed that Ahsan *et al.* proposed a similar approach to check semantic equivalence of IP prefix tables [75]. However, their scheme needs to do normalization and leaf pushing for two routing tables to build two complete binary trees. Moreover, the two binary trees have to mutually check each other for the

longest prefix using a complicated process and handle comparisons for different scenarios. We use a more efficient and straightforward way to accomplish the same task. Specifically speaking, one complete binary tree (each node either has no child or two children) and two Patricia Trie structures will be used.

Firstly, one complete binary tree is built to generate covering prefixes for the whole IP space based on the union of prefixes from the two routing tables, and each Patricia Trie stores information of one exact routing table including prefixes and next-hops. Secondly, the algorithm picks the prefix from each leaf node on the complete binary tree, and uses it to lookup the corresponding next-hop on the two Patricia Tries with the longest prefix match rule. Thirdly, it does a comparison between the two next-hops, if they are the same for all prefixes coming from the complete binary tree, then the two routing tables have the same forwarding behavior, and the forwarding correctness is guaranteed. Otherwise, they have different forwarding behaviors. The approach will always pick the longest prefix and cover all of the IP space holes the two routing tables have. Also it will generate the same results as the brute-force way to search the entire IP space one by one ranging from 0.0.0.0-255.255.255.255.

We proved the correctness of the algorithm and omit it here due to space constraint. This algorithm eliminates massive computational overhead compared with the brute-force way. For example, the original routing table without aggregation and incremental update handling has 378,668 entries, and the aggregated routing table has 179,993 entries. The built binary tree has 577,253 leaf nodes all together, thus we only need to do 577,253*2 = 1154,506 lookups on the two Patricia Tries to obtain the final results. However, the brute-force way will use about 4 billion IP addresses to do 4*2=8 billion lookups to achieve the same goal. We improved the verification performance speed by 7440.355 times.

4 Implementation

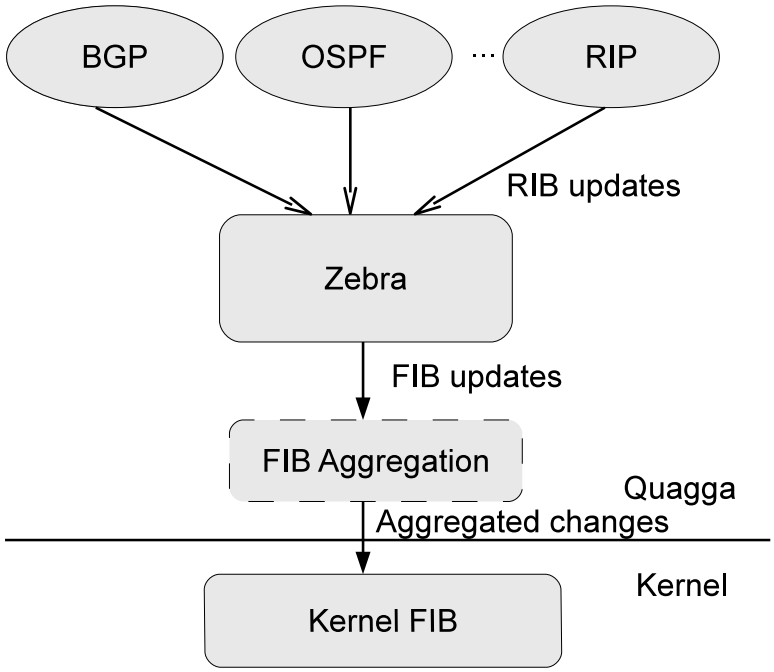


Figure 3.13: FIB Aggregation in Quagga

From the offline analysis results, we can see that FIFA is able to do handle incremental FIB updates quickly while still maintaining very good compression ratios. At the meeting of NANOG 56 [4] in October 2012, many network operators/vendors, such as Brocade, Juniper, and Cisco, expressed their interest in our FIB aggregation results. However, they also expressed their concerns regarding the processing overhead under their realistic network environment. For example, what is the aggregation ratio when applied to a FIB using IGP next-hops and what is the implementation cost to incorporate the algorithm into their own router software suite. People in general agree that the idea is great, but are uncertain about how many benefits it can bring compared with the costs.

Also we obtained feedback from other conversations and concluded that FIB aggregation has great commercial potential, but its effectiveness needs to be

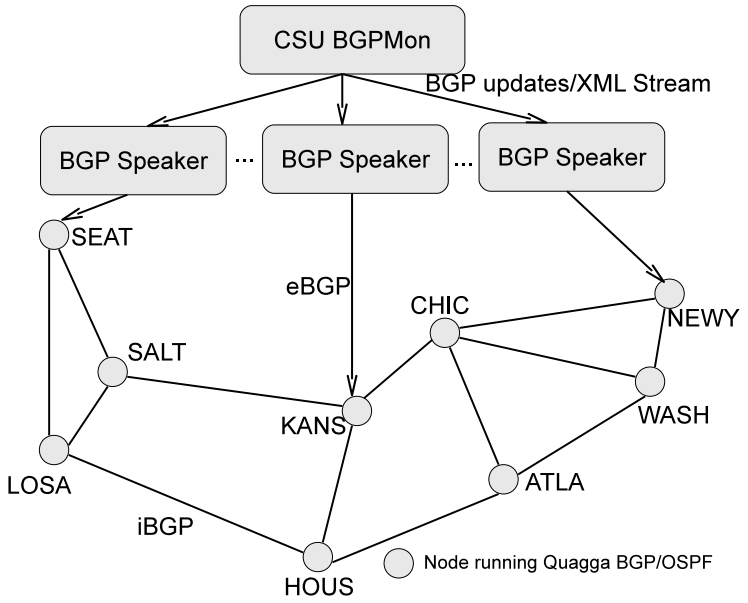


Figure 3.14: Real-time FIB Aggregation Architecture Design

demonstrated and verified quantitatively under realistic network environment. Therefore, we propose to build a real-time demonstration system of FIB aggregation as the next step towards the realization of its commercial potential. First, we implemented FIFA algorithms in Quagga [13], a widely used open-source routing software suite. This demonstrated the feasibility of FIFA. Second, we used a dedicated server to build a virtual network. The server ran multiple virtual machines, with each virtual machine configured as a router running an instance of Quagga. These virtual routers are connected to form a virtual network. The virtual network will have the same topology as a real network such as Internet2 and can be reconfigured easily. Third, we received real-time routing updates from Colorado State Universitys BGPMon project [51], and fed them into the virtual routers which run regular routing protocols and perform FIB aggregation at the same time. Through this system, we can monitor FIB size and CPU overhead of the virtual routers, which demonstrate the effectiveness and performance of FIFA algorithms under realistic network environment.

Ideally, our FIB aggregation algorithm should be implemented on a commercial router to test its effectiveness with the processing overhead. However, commercial router software suites such as from Cisco [2] or Juniper [7], are not open source due to business confidentiality and proprietary issue. Fortunately, Quagga is a widely used open-source routing software suite and architecturally similar to many proprietary router implementations. A system installed with Quagga acts as and has all the TCP/IP functionalities of a dedicated router. A machine with Quagga can exchange routing information with other machines with Quagga or other normal routers using routing protocols, such as BGP, OSPF and so forth.

Quagga [13] was made from a collection of several daemons that work together to build the routing table, which is different from traditional routing softwares which use one process program to provide all routing protocol functionalities. More specifically, in Quagga, all routing protocols such as OSPF, RIB, BGP, and the like, are implemented as daemons and these daemons exchange routing information with the Zebra daemon, which is responsible for maintaining and communicating with the kernel table, namely, FIB. Actually, the Zebra daemon uses a netlink socket to communicate with the kernel table. If there is a new route or a route needs to be updated, then the function rib_install_kernel() will be called. Otherwise, if there is a withdrawal message, then the function rib_uninstall_kernel() will be called. In our design, we first intercept the update messages from Zebra and feed these messages to our FIB aggregation module. Second, our FIB aggregation module handles the updates using FIFA algorithms and output the necessary messages to Zebra. At this point, Zebra calls the two functions for kernel table updates or withdrawals according to the returned message types. Figure 3.13 illustrates the FIB aggregation module in Quagga.

In order to show the effectiveness of our FIB aggregation, we utilized real-time BGP routing information. Different from other existing BGP monitoring software

such as Zebra and Quagga, BGP Monitoring System (BGPMon) [51] was designed to monitor BGP updates and routing tables from BGP routers in real time. Therefore, we can take advantage of BGPMon to access real-time BGP data. However, BGPMon only provides live BGP data stream in XML format and cannot serve as a BGP speaker to announce the routes to other BGP peers. This fact introduces challenges in applying real-time BGP updates to a BGP table implemented in Quagga. Therefore, we have to find or implement a customized BGP speaker which can read data from the BGPMon XML stream, convert the data to BGP messages, and announce the BGP messages to its peers. Figure 3.14 illustrates the real-time FIB aggregation demonstration system architectural design.

After careful investigation, we found BGPSimple [1], a simple customized BGP speaker which was implemented in Perl and can set up BGP adjacency with a BGP peer. BGPSimple is also able to monitor messages and updates received from the peer and send updates from a predefined set of NLRIs/attributes. In our experiment, we can feed the converted BGP messages from BGPMon XML stream to BGPSimple. However, we have made some modifications to BGPSimple. First, the original BGPSimple reads all predefined BGP messages from a local file and we changed it to read data from a pipeline. Also, the BGPSimple does not implement BGP withdrawal functionality and we adapted it to work for BGP withdrawal messages.

The input format for BGPSimple is TABLE_DUMP_V2 format [16], this means we have to convert the BGP XML messages to individual TABLE_DUMP_V2 messages. Thanks to the BGPMon team, they have developed some tools to convert XML messages to TABLE_DUMP_V2 messages. However, their tool does not convert all BGP message attributes so we added the necessary ones for our use and we are now able to obtain real-time TABLE_DUMP_V2 messages. After feeding them into the customized BGPSimple speaker, we are able to inject and withdraw

routes from the routing table maintained by Quagga.

Finally, after connecting all the required components together, we ran a few Quagga instances with aggregation functionalities and others which did not have the aggregation functionalities. Through comparing their aggregation results, we demonstrated that our FIB aggregation schemes could be hooked up to existing router operating systems with small overhead.

Chapter 4

FIB Caching

1 Design Overview

Figure 4.1 illustrates a router architecture with the proposed FIB caching scheme. The control plane contains the RIB, while the Slow FIB and Cache reside in the data plane. The Slow FIB memory contains a copy of the full forwarding table with all prefix entries and next-hop information. The Cache contains only the most popular prefixes driven by data traffic. We place the Slow FIB in the data plane (in the line card) rather than the control plane (in the route processor) so that a cache miss/replacement can be quickly handled. The Slow FIB handles four events A, W, M and O representing Route Announcement, Route Withdrawal, Cache Miss and Cache Replacement, respectively. The Route Announcement and Route Withdrawal events are generated as a result of RIB updates, which need to be propagated to the FIB. Cache Miss and Cache Replacement are events from the cache. The former happens when an incoming packet does not have a matching prefix in the Cache. The latter occurs when the Cache is full. In the remainder of this paper, full FIB or FIB refers to the Slow FIB, and operations occur in the Slow FIB unless the location is explicitly stated. Before discussing the operations that take place in the Slow FIB and Cache, we explain the cache-hiding problem and outline our solution for handling it in the following section.

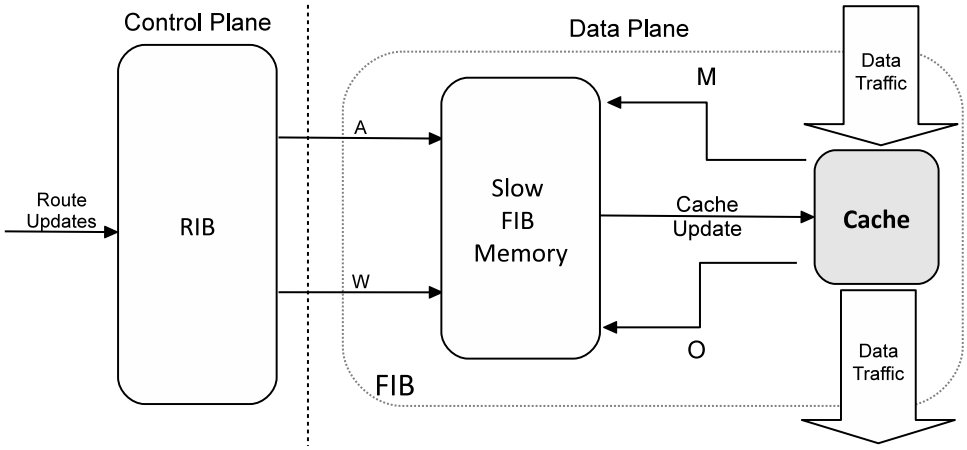


Figure 4.1: Design Architecture for FIB Caching

1.1 Cache-Hiding Problem

FIB caching is different from traditional caching mechanisms – even if a packet has a matching prefix in the cache, it may not be the correct entry for forwarding the packet if there is a longer matching prefix in the full FIB. Below we use a simple example to illustrate this cache-hiding problem. For ease of illustration, we use 8-bit addresses and binary representations of addresses in our examples.

Suppose a FIB table contains three prefixes as shown in Table 4.1(a), and the corresponding cache is empty (not shown). Assume a data packet destined to 10011000 arrives at a router. The router then looks for the longest prefix match in the cache, which has no matching entry (the cache is empty). The router then looks up the full FIB in slow memory and loads the matching entry 1001/4 with the next-hop 2 to the cache (Table 4.1(b)). Now, suppose another data packet destined to 10010001 arrives; the router will first check the cache to see if there is a prefix matching the destination IP address. It finds the matching prefix 1001/4 in the cache and thereby sends the packet to next-hop 2. This is, however, incorrect because the real matching prefix for IP address 10010001 should be the more specific prefix 100100/6 with the next-hop 1. In other words, the cached prefix

Table 4.1: FIB entries and Cache Entries (The cache is initially empty and it receives one entry upon the first cache miss.)

(a) FIB Entries

Label	Prefix	Next Hop
A	10/2	4
B	1001/4	2
C	100100/6	1

(b) Cache Entries

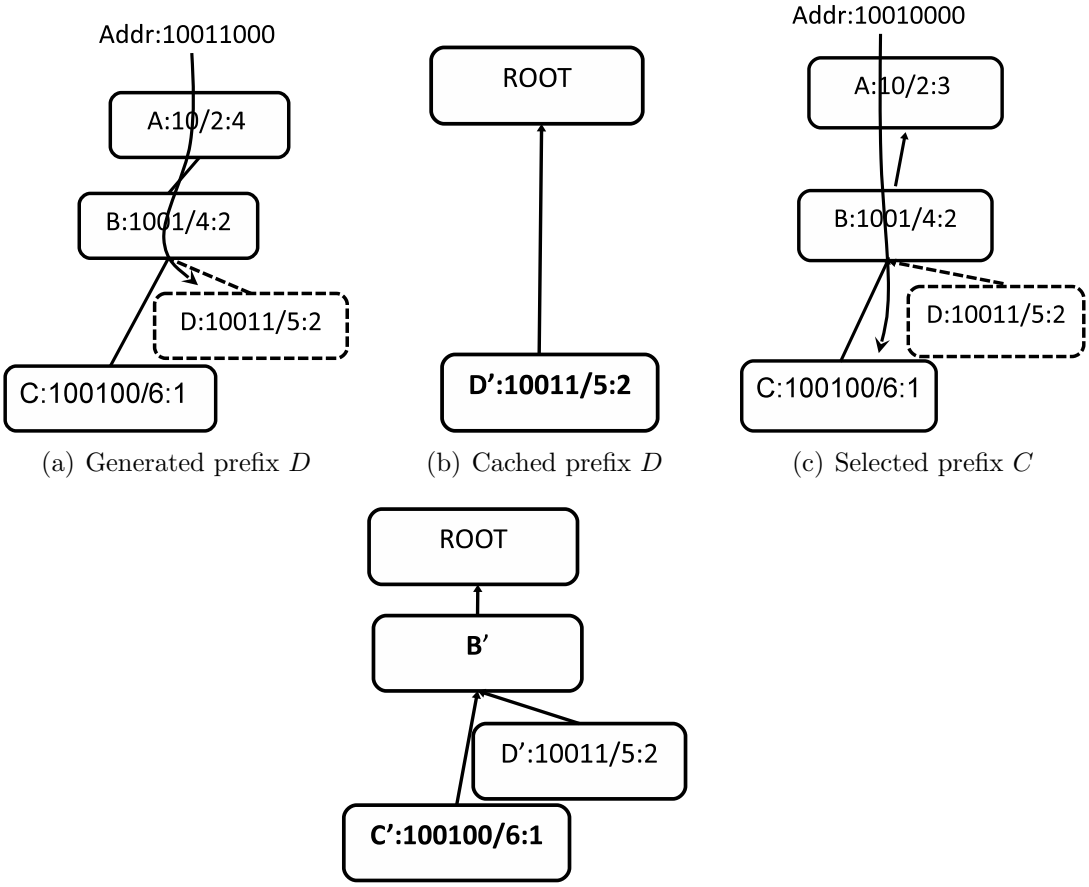
Label	Prefix	Next Hop
B	1001/4	2

1001/4 "hides" the more specific prefix 100100/6 in the full FIB.

1.2 Our Solution to Cache-Hiding

To illustrate our solution, we use Patricia Tries (*i.e.*, Radix Tree) [58] to store the slow FIB and cached prefixes. A Patricia Trie is a space-optimized tree where the child prefix can be longer than the parent prefix by more than one. It is commonly used to store routing tables in a compact manner. Note, however, that our solution can be applied to any tree based structures.

We cache the most specific *non-overlapping* prefixes that do not hide any longer prefixes in the full FIB to avoid the cache hiding problem. In Table 4.1(a), C's address space is covered by B, so they are *not* non-overlapping prefixes (see Figure 4.2(a)). As such, we cannot simply load the prefix B (1001/4) into the cache, because it will cause a problem for the next packet destined to the address 10010000. Instead, we need to generate a leaf prefix D (10011/5) to represent the address space under B that does not overlap with C (Figure 4.2(a)) and put it into the cache (Figure 4.2(b)). D (10011/5) has the next-hop 2, same as its covering prefix B (1001/4). The next packet destined to 10010000 causes a cache miss again and correctly finds a matching prefix C (100100/6) with the next-hop 1 in the slow FIB (Figure 4.2(c)), which is then loaded into the cache (Figure 4.2(d)). We call our approach *FIB Caching using Minimal Non-overlapping Prefixes* because we select or generate only the shortest leaf prefixes needed by the data traffic to minimize the



(a) Generated prefix D (b) Cached prefix D (c) Selected prefix C

(d) Cached prefix C

Figure 4.2: Selection or generation of a leaf prefix

number of cached prefixes.

1.3 Slow FIB Operations

Upon receiving an *announcement* or *withdrawal* event from the RIB, the slow FIB updates the corresponding entry and updates the cache if necessary. The specific operations to update the cache are described in Section 2.5 and 2.6.

Upon a *cache miss* event, the FIB returns to the cache a leaf prefix that matches the data packet that caused the cache miss. A new leaf prefix may need to be dynamically generated if the existing leaf prefixes in the FIB do not match the data packet (Section 2.3). Upon receiving a *cache replacement* message, the FIB will

either delete or update an entry according to different scenarios which will be discussed in Section 2.3 and 2.4 in detail.

1.4 Cache Operations

Cache operations include cache initialization, cache-miss traffic handling, and cache replacement.

Cache Initialization. Handling the initial traffic is a major concern for deploying cache mechanisms [83]. To address this issue effectively, we fill up the initial empty cache with a set of leaf prefixes from the FIB that cover the most IP addresses. More specifically, breadth-first search is employed to find the shortest leaf prefixes from the slow FIB Trie (up to the cache size). This way we achieve a high hit ratio while avoiding the cache-hiding problem.

Cache Miss Traffic Handling. Packets experiencing cache misses can be stored in a separate queue and forwarded once the prefixes from slow FIB memory are installed into the cache.

Cache Replacement. We use the LRU (Least Recently Used) replacement algorithm when a new prefix needs to be installed into a full cache. Our decision is based on the study conducted by Kim *et al.* [46], which shows that the LRU algorithm performs almost as well as the optimal cache replacement algorithm.

2 Design Description

2.1 Workflow for Handling Data Traffic

Figure 4.3 shows how our cache handles an incoming packet. In the 'Init' or initialization phase, we load all FIB entries into the slow FIB . Subsequently, we fill up the entire cache with the leaf prefixes that have the shortest length. For any

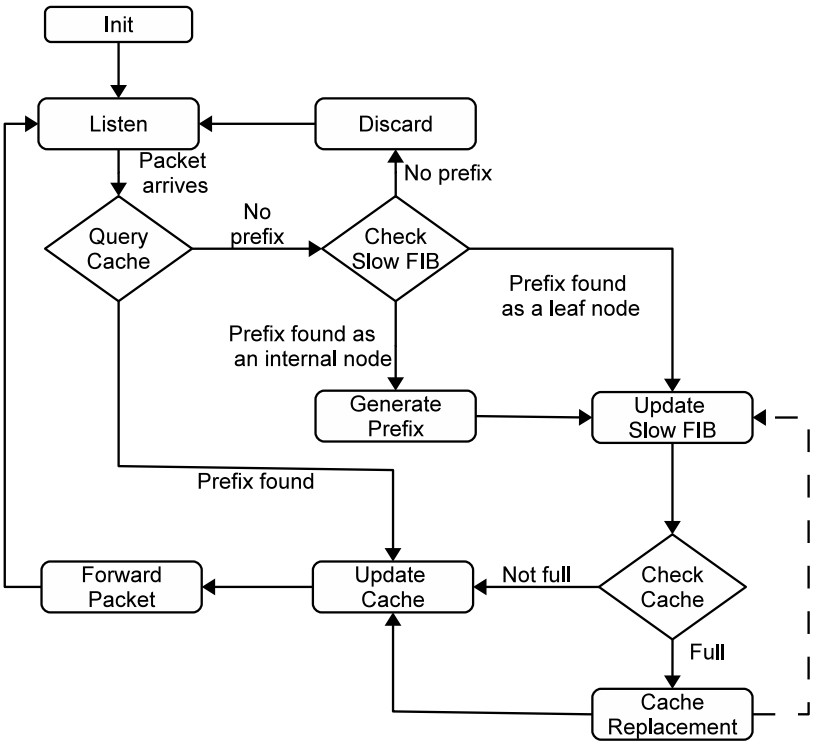


Figure 4.3: Workflow for Handling Incoming Data Traffic (the dotted line means that during Cache Replacement, the slow FIB needs to be updated but the flow of operation does not continue beyond that point.)

incoming packet, a longest prefix match is performed on the Patricia Trie of the cache. In the case of a prefix match, the packet is forwarded accordingly and the prefix's status becomes the "latest accessed" to facilitate the cache replacement policy later.

In the case of a *cache miss*, a lookup is performed in the slow FIB and the packet is discarded if the lookup returns no matching prefix. On the other hand, if the longest matching prefix is a leaf node in the Patricia Trie, it is pushed to the cache. Otherwise, i.e., the prefix is an internal node, a more specific prefix is created and pushed to the cache (Section 3.3). The packet is then forwarded to the corresponding next-hop. When pushing any prefix to a *full* cache, the cache replacement mechanism removes the least recently used prefix from the cache and installs the new one.

2.2 Data Structure

Each node in the Patricia Trie of the slow FIB is one of four types, which may change upon an update. These types help us keep the cache, slow FIB and the RIB consistent with each other. The four types are as follows (note that this classification does not apply to the cache): (a) $CACHE\_ONLY$: a *leaf* node that is created on demand as a result of the cache miss event; (b) FIB_ONLY: a node derived from the original routing table or RIB update, but the prefix is *not* in the cache; (c) FIB_CACHE: a leaf node derived from the routing table and the prefix is in the cache; and (d) $GLUE\_NODE$: any other auxiliary node except the above three types.

2.3 Handling Cache Misses

In the case of a cache miss, we perform a longest prefix matching in the slow FIB and may encounter the following three cases: (1) if there is no matching node, then drop the packet; (2) if there is a matching leaf node with the type FIB_ONLY, then set the type to FIB_CACHE, and install the prefix with the corresponding next-hop into the cache; and (3) if there is a matching internal node with the type FIB_ONLY, generate a $CACHE\_ONLY$ node as described below and install it into the cache.

Suppose P_L and P_R are the left and right child of a node P, respectively, and X is the destination IP address. We generate a $CACHE\_ONLY$ node with the same next-hop as its parent on the trie and a prefix containing the first $l + 1$ bits of X, where l is defined as follows: (a) if P_L is NULL, then compare P_R with X to get the common portion Y with length l; (b) if P_R is NULL, then compare P_L with X to get the common portion Y with length l; and (c) if P_L and P_R are not NULL, compare X with P_L and P_R separately, and get the common portion Y_L and Y_R, then find the

longer prefix Y with length l from Y_L and Y_R.

Now we provide a detailed example of Case c mentioned above. In Figure 4.4(a), the matching node B with prefix 1001/4 has both a left child (C) and a right child (D). So $Y_L = (X \ \& \ C) = (10010100 \ \& \ 100100/6) = 10010/5$ and $Y_R = (X \ \& \ D) = (10010100 \ \& \ 10011/5) = 1001/4$. Therefore, we pick the longer one, Y=Y_L=10010/5 and l=5. The prefix of the imaginary parent or glue node (E) is 10010/5 as shown in Figure 4.4(b) and the new leaf node (F) is $X/(l+1)$=100101/6 as shown in Figure 4.4(c). F's node type is $CACHE\_ONLY$, as it is generated on demand and will be installed into the cache. Figure 4.4(d) and 4.4(e) show the cache entries before and after the update.

2.4 Handling Cache Replacement

When the cache becomes full, some prefixes in the FIB cache need to be evicted according to the LRU replacement strategy. We first remove the prefix from the cache and then update the slow FIB. Two cases may happen:

(1) If the corresponding node type is $CACHE\_ONLY$, it means that the node was created on-demand and there would be no such entry in the RIB, so we can remove it directly.

(2) If the corresponding node type is FIB_CACHE, it means that this node was originally from the RIB or RIB update, so we cannot remove it completely from the FIB. Therefore, we change the type to FIB_ONLY.

Figure 4.5 shows a Cache Replacement event on prefix 100100/6 (the FIB_CACHE case). Figure 4.5(a) and 4.5(b) show the cache operations. Figure 4.5(c) and 4.5(d) show the slow FIB operations.

2.5 Handling Route Announcements

In this section and Section 2.6, we discuss the handling of route announcements and withdrawals, respectively (the complete workflow for both is depicted in Figure 4.6). A route announcement may add a new prefix or update an existing entry in the slow FIB. Below we describe each scenario in detail.

When adding a new node to the FIB trie, we need to handle the following two cases.

1. The new node is added as a leaf node: if its direct parent node type is $CACHE\_ONLY$ (i.e., the prefix of this node was generated on demand and is in the cache), then we remove the parent node from both the FIB and the cache, in order to avoid the cache-hiding problem. If the direct parent of the new node is a FIB_ONLY, nothing needs to be done, because the parent node must not be in the cache. If the direct parent of the new node is FIB_CACHE (i.e., the prefix attached to the parent node is in the cache, and needs to be removed from there), then we set the parent node type as FIB_ONLY and remove the prefix from the cache.

2. The new node is added as an internal node: all the $CACHE\_ONLY$ nodes whose next-hops are derived from this node should have the same next-hop as this one, so we update these nodes with the new next-hop and update the cache accordingly to synchronize the next-hop information. We do not update the FIB_CACHE nodes because their next-hops are directly from the corresponding real prefixes in the RIB, not derived from their ancestors.

Similarly, we need to handle two cases when updating an existing FIB entry to change its next hop value (see below).

1. The existing node is a leaf node: if the node type is FIB_ONLY (i.e., the prefix is not in the cache), we simply update it with the new next-hop.

Otherwise, we update it with the new next-hop, set its type as FIB_CACHE, and update its next-hop in the cache accordingly.

2. The existing node is an internal node: we update all the $CACHE\_ONLY$ nodes whose next-hops are derived from this node with the new next-hop, and update the cache accordingly.

Figure 4.7 depicts an Announcement example, in which an update with the next-hop 3 comes for an existing node (E) with the prefix 10010/5. As the update is for an internal node, the node type will be changed to FIB_ONLY. Moreover, we update those $CACHE\_ONLY$ nodes whose next-hops are derived from this node, in order to keep the forwarding behavior correct. For example, in Figure 4.7(a), node F with the prefix 100101/6 is a $CACHE\_ONLY$ node that inherited its next-hop from E, so its next-hop is changed to 3 in Figure 4.7(b), and we update the corresponding entry in the cache as shown in Figure 4.7(c) and Figure 4.7(d). As a subsequent, the data packet destined to 100101/6 will be forwarded correctly to the new next-hop 3.

2.6 Handling Route Withdrawals

For a route withdrawal, we need to process it only if it matches an existing node in the FIB, since the corresponding prefix is supposed to be in the FIB in order to be "withdrawn". The matching node can be either a leaf node or an internal node, and we process it as follows. (1) Leaf node: If the node type is FIB_CACHE, we delete it from both the FIB and the cache. If the node type is FIB_ONLY, we delete it from the FIB only, since it is not in the cache.

(2) Internal node: we delete its next-hop and change the node type to $GLUE\_NODE$ (it is still useful in the FIB trie to connect the other nodes). Since our algorithm puts only leaf nodes in the cache, this internal node cannot be in the

cache and therefore no cache update is needed. Then, we update the *next-hop* field of those $CACHE\_ONLY$ nodes whose next-hops are derived from this node. Finally, we update the cache accordingly.

Figure 4.8 shows an example of removing the prefix 1001/4 (the internal node case). First, the type of node B (1001/4) is set to $GLUE\_NODE$. The next step is to update the affected $CACHE\_ONLY$ nodes. In this example, node D (10011/5) is the only affected $CACHE\_ONLY$ node. It is updated with the next-hop (4) of its nearest ancestor A as shown in Figure 4.8(b). Subsequently, the cache entry $D\prime$ with prefix 10011/5 is updated with the new next-hop 4 as shown in Figure 4.8(d).

2.7 Pseudo Code

Algorithm 9 contains the main functions to perform FIB and cache update upon any update message, including cache miss and age-out handling. There are four types of updates: A, W, M, and O for Announcement, Withdrawal, Cache Miss and Age Out, respectively. They are handled separately through different type values. $Message$ contains both prefix and next-hop information if the update type is an announcement; otherwise, it only contains prefix information if the update type is withdrawal or an age-out. The function $LookUp$, given a prefix, conducts lookup operations according to the longest prefix match rule (LPM), and returns the node if found, otherwise, it returns $NULL$. The function $MakeNode$ adds a node with type FIB_ONLY to a trie given a prefix and next-hop. Given the corresponding prefix from a trie, the function $RemoveNode$ removes a node. The function $UpdateCache$ updates the corresponding node value in the cache given a prefix and next-hop information. The function $UpdateSubtree$ recursively goes through a subtree of the given node in the slow FIB trie to find all nodes with type $CACHE\_ONLY$, and updates the cache counterpart with the given next-hop value. Notice that the function $CacheMissUpdate$ uses an IP address as one of its

arguments. It is the destination IP address for the data packet on which the cache miss occurs. The function needs to use the address to generate on-demand the leaf node with type $CACHE\_ONLY$ in the slow FIB memory and install it into the cache. The new prefix generation process has been described in detail in Section 2.3.

Procedure 9 $UPDATE(FIB, Cache, Message, Type)$

```
1: if Type = A(Announcement) then
2:     AnnouncementUpdate(FIB, Cache, Message)
3: else if Type = W(Withdrawal) then
4:     WithdrawalUpdate(FIB, Cache, Message)
5: else if Type = M(CacheMiss) then
6:     CacheMissUpdate(FIB, Cache, IPAddress)
7: else if Type = O(AgeOut) then
8:     AgeOutUpdate(FIB, Cache, Message)
```

Procedure 10 $AnnouncementUpdate(FIB, Cache, Message)$

```
1:  P ← Message.prefix
2:  N ← Message.next − hop
3:  node ← LookUp(FIB, P)
4:  if node = NULL then
5:     node ← MakeNode(FIB, P, N)
6:     if node.l = NULL&node.r = NULL then
7:        if node.parent.type = CACHE_ONLY then
8:           RemoveNode(FIB, node.parent.prefix)
9:           RemoveNode(Cache, node.parent.prefix)
10:       else if node.parent.type = FIB_CACHE then
11:          node.parent.type ← FIB_ONLY
12:          RemoveNode(Cache, node.parent.prefix)
13:    else
14:       UpdateSubtree(FIB, Cache, P, N)
15: else
16:    node.next − hop ← N
17:    if node.l = NULL&node.r = NULL then
18:       if node.type = CACHE_ONLY then
19:          node.type ← FIB_CACHE
20:          UpdateCache(Cache, P, N)
21:       else if node.parent.type = FIB_CACHE then
22:          UpdateCache(Cache, P, N)
23:    else
24:       node.type ← FIB_ONLY
25:       UpdateSubtree(FIB, Cache, P, N)
```

Procedure 11 $WithdrawalUpdate(FIB, Cache, Message)$

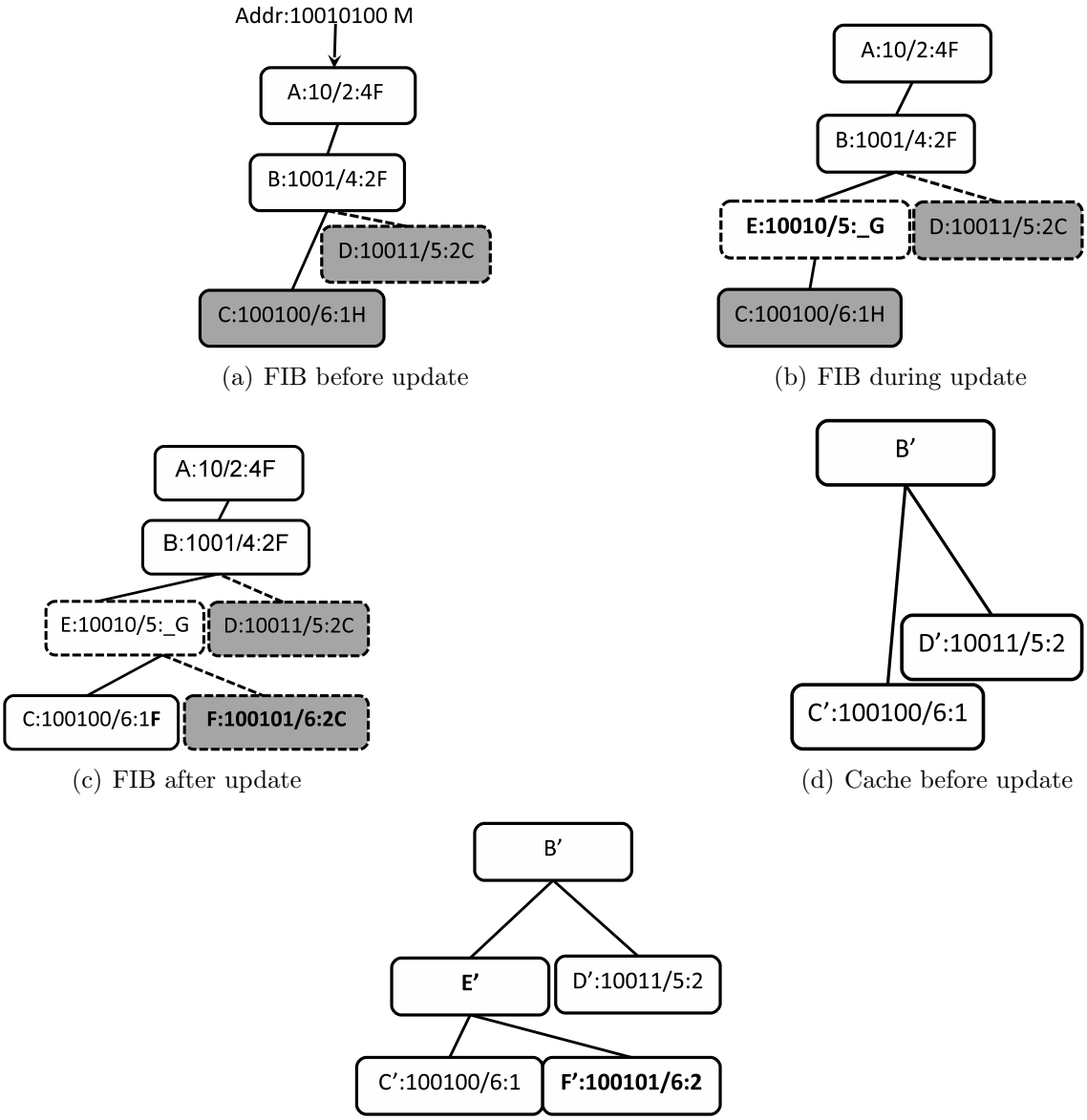
```
node ← LookUp(FIB, Message.prefix)
if node! = NULL then
    ancestor ← getAncestor(node)
    if node.l = NULL&node.r = NULL then
        if node.type = FIB_CACHE then
            node.type ← CACHE_ONLY
            node.next − hop ← ancestor.next − hop
            P ← node.prefix
            N ← node.next − hop
            UpdateCache(Cache, P, N)
        else if node.type = FIB_ONLY then
            RemoveNode(FIB, Message.prefix)
    else
        P ← node.prefix
        N ← ancestor.next − hop
        UpdateSubtree(FIB, Cache, P, N)
        RemoveNode(FIB, Message.prefix)
```

Procedure 12 $AgeOutUpdate(FIB, Cache, Message)$

```
RemoveNode(Cache, Message.prefix)
node = LookUp(FIB, Message.prefix)
if node ≠ NULL then
    if node.type = CACHE_ONLY then
        RemoveNode(FIB, node.prefix)
    else if node.type = FIB_CACHE then
        node.type = FIB_ONLY
```

Procedure 13 $CacheMissUpdate(FIB, Cache, IPAddress)$

```
1:  node = LookUp(FIB, IPAddress)
2:  if node ≠ NULL then
3:      P ← node.prefix
4:      N ← node.next − hop
5:      if node.l = NULL&node.r = NULL then
6:          node.type ← FIB_CACHE
7:          updateCache(Cache, P, N)
8:      else
9:          if node.l = NULL then
10:             C ← P ∧ node.r
11:         else if node.r = NULL then
12:             C ← P ∧ node.l
13:         else
14:             C_1 ← IPAddress ∧ node.l
15:             C_2 ← IPAddress ∧ node.r
16:             C ← (C_1.length > C_2.length)?C_1 : C_2
17:         P_x = IPAddress/C.length
18:         nd ← GenerateLeafNode(FIB, P_x)
19:         nd.type = CACHE_ONLY
20:         updateCache(Cache, nd.prefix, N)
```



(a) FIB before update

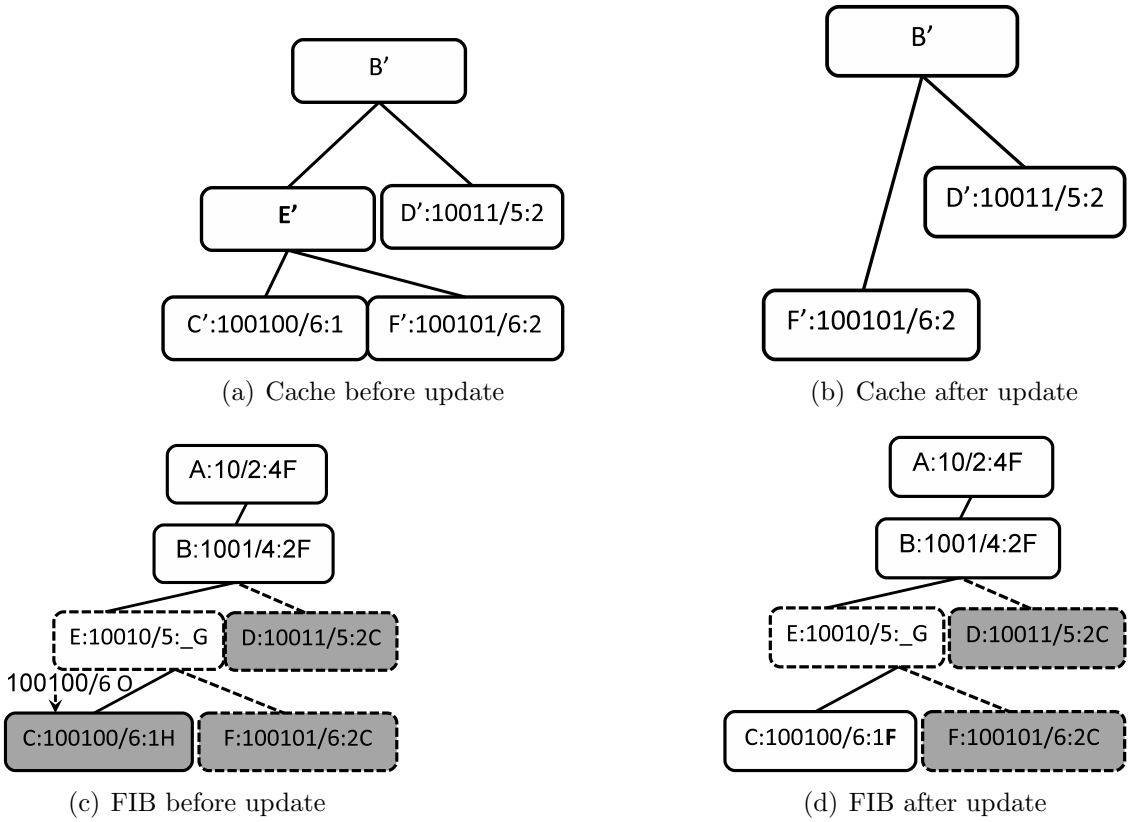
(b) FIB during update

(c) FIB after update

(d) Cache before update

(e) Cache after update

Figure 4.4: Example of Cache Miss Update. There are three fields for each node from left to right: prefix, next-hop and node type (F: FIB_ONLY, H: FIB_CACHE, C: CACHE_ONLY and G: GLUE_NODE) in the FIB trie. A bold font denotes a field updated in the current step. A solid rectangle denotes a node with a prefix from the original routing table or an update. A dashed rectangle denotes a generated node due to cache miss update. A grey node denotes a node in the cache.



(a) Cache before update

(b) Cache after update

(c) FIB before update

(d) FIB after update

Figure 4.5: Example of Cache Replacement Update

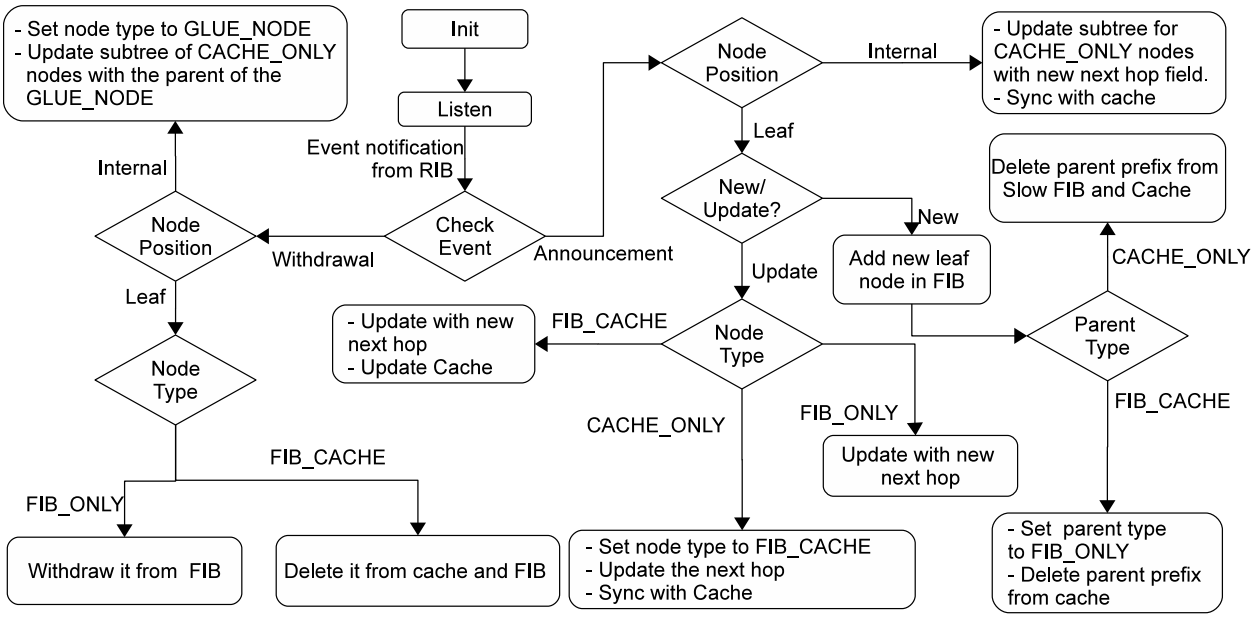
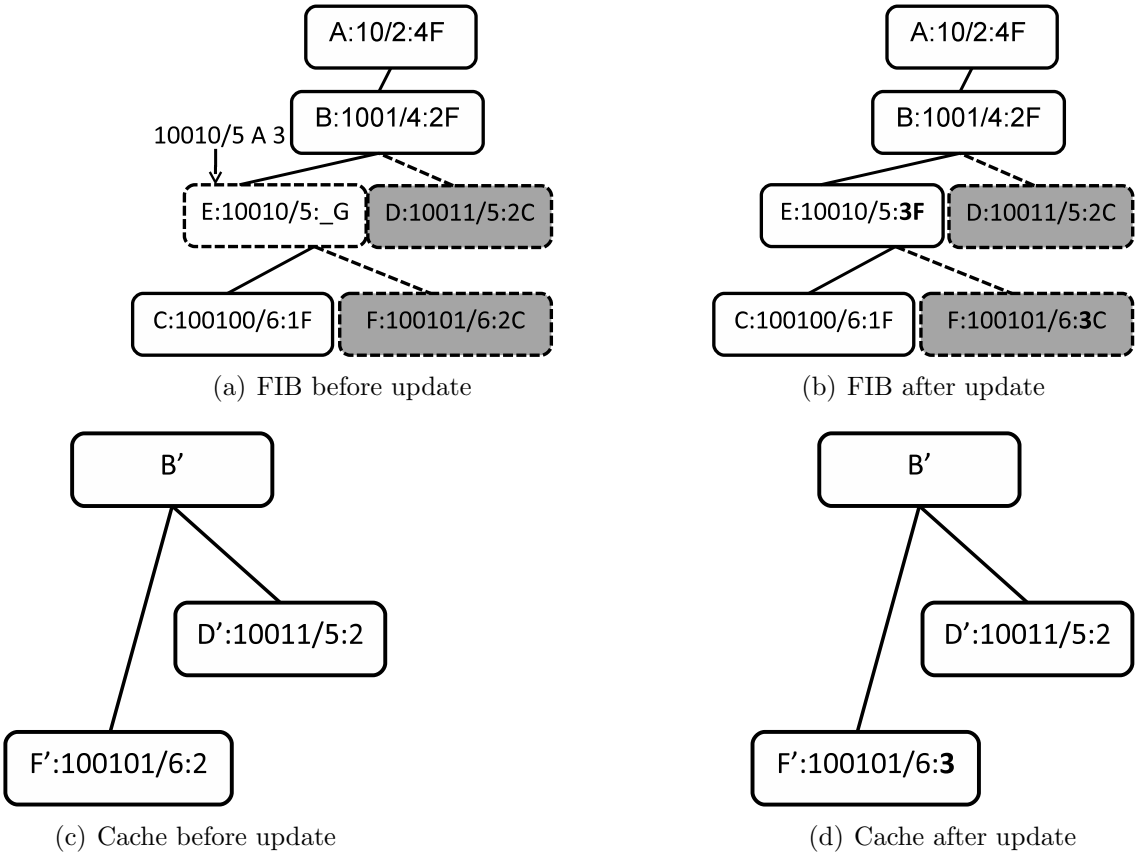


Figure 4.6: Workflow for Handling Announcements and Withdrawals (loopbacks to the 'Listen' state are not shown.)



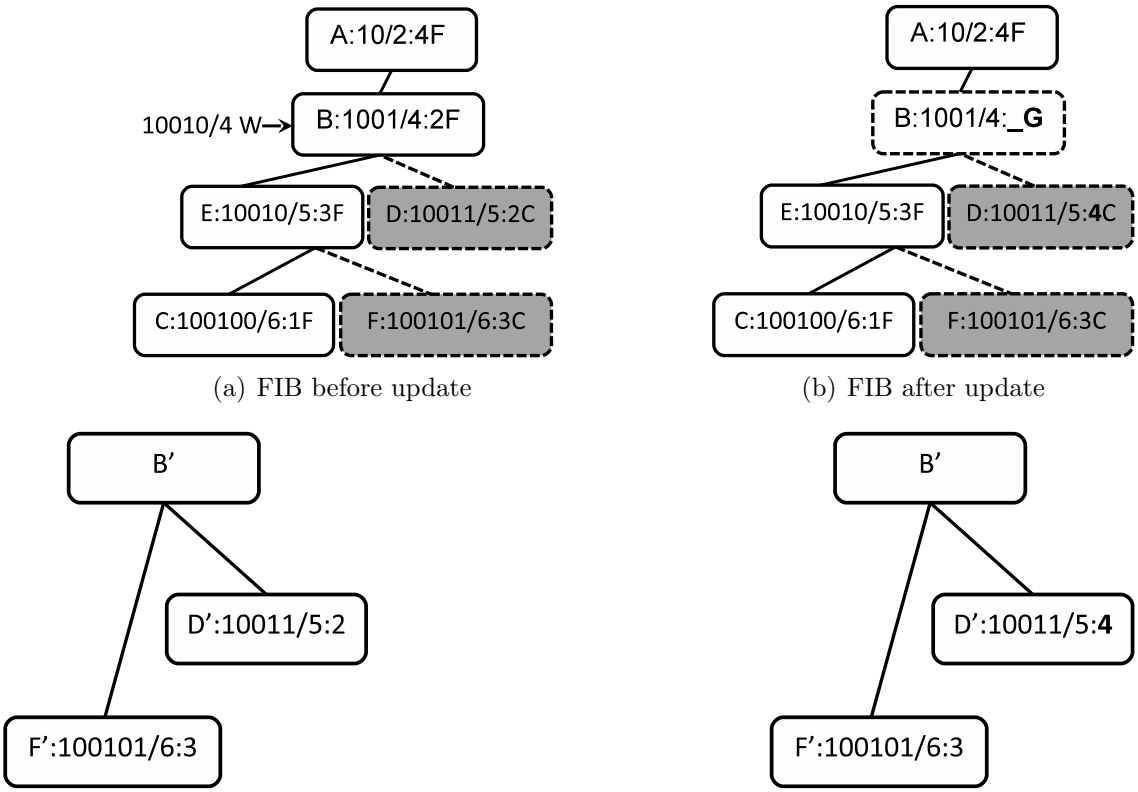
(a) FIB before update

(b) FIB after update

(c) Cache before update

(d) Cache after update

Figure 4.7: Example of Announcement Update



(a) FIB before update

(b) FIB after update

(c) Cache before update

(d) Cache after update

Figure 4.8: Example of Withdrawal Update

3 Evaluation

To evaluate our scheme, we used a 24-hour traffic trace of more than 4.1 billion packets from a regional ISP collected from 12/16/2011 to 12/17/2011. We obtained routing tables and updates of 30 different routers from the route-views2 data archive [18] on 12/16/2011 and 12/17/2011. After the initialization of the slow FIB and cache, we run our caching scheme with the data and updates. The updates and data are also passed through an emulated router without the cache to verify the forwarding correctness of our scheme. Our results are similar for all the 30 routers, so we present the results from only one of them in most cases.

3.1 Traffic Distribution

Figure 4.9 shows the traffic distribution over the prefixes from the forwarding table of one of the thirty routers. The x-axis represents the popular prefix rank, and the y-axis represents the cumulative percentage of the IP packets covered by the popular prefixes. We make two main observations: (a) the top 10, 100, 1K, 10K, 20K popular prefixes cover about 42.79%, 79.18%, 93.81%, 99.51%, and 99.87%, respectively, of the traffic, which supports a common finding from several other studies [37, 46, 67, 81], i.e., a very small number of popular prefixes contribute to most of the traffic; and (b) most of the entries in the global routing tables are not in use during this period. In fact, more than 70.18% of the FIB entries were not used at all, which further suggests the feasibility of introducing an efficient caching mechanism for the routers.

3.2 Hit Ratio

The *hit ratio* of a cache is the success rate of finding a matching entry in the cache. It is considered one of the most important metrics for evaluating a caching scheme.

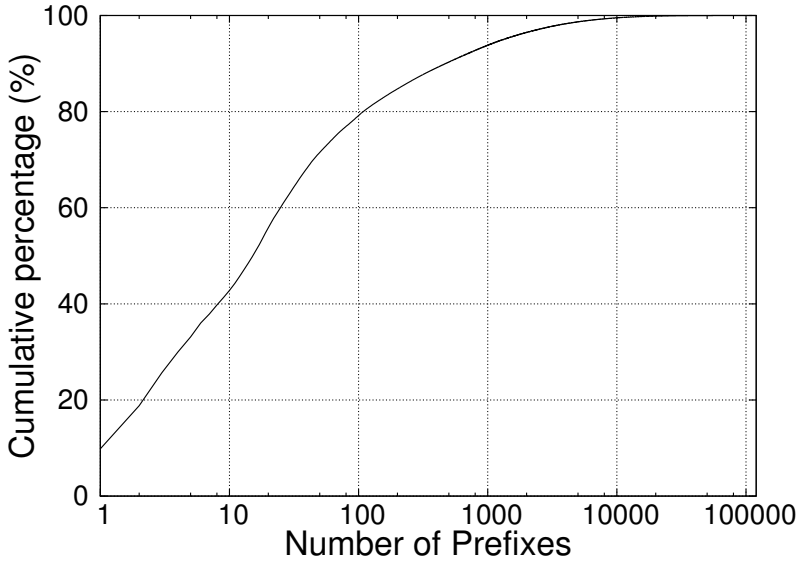
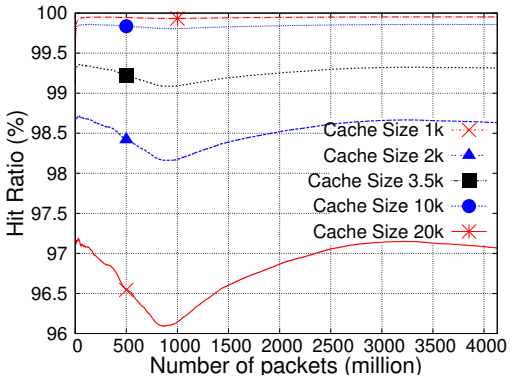


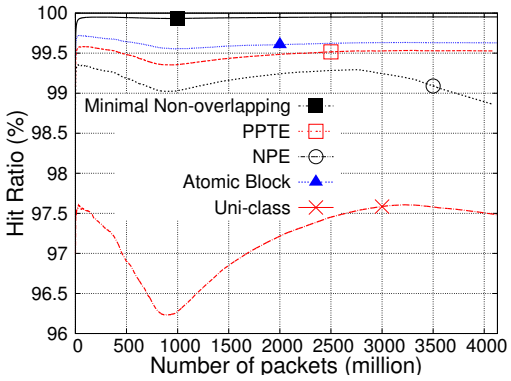
Figure 4.9: Traffic Distribution On Non-overlapping Prefixes

For a given cache size, the higher the hit ratio is, the better the cache scheme would be. In our experiments, we obtain the hit ratios for 30 routers with different cache sizes ranging from 1K to 20K prefixes. Figure 4.10(a) shows different hit ratios for one router with five different cache sizes over the 24-hour period. We observe that on average the hit ratio is 96.83%, 98.52%, 99.25%, 99.84%, 99.95% for the cache size of 1K, 2K, 3.5K, 10K and 20K, respectively. The dips around 870 million data packets are due to the traffic pattern around 7:30am, which has the lowest traffic rate but a similar number of distinct destination addresses. This leads to a high miss ratio, as we are dividing roughly the same number of cache misses with a much lower number of packets. Furthermore, we found that the hit ratio tends to be more stable when cache size increases. Other routers have very similar results to this one.

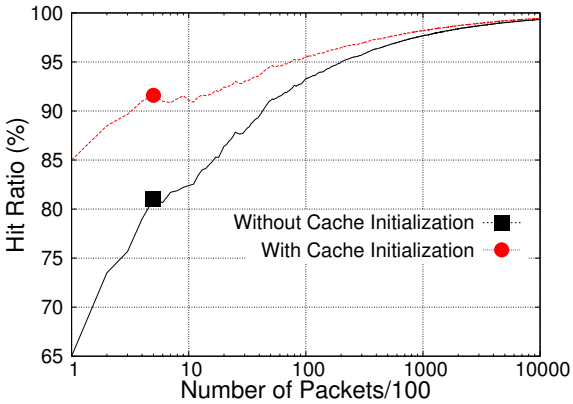
We also compared our scheme with different cache-hiding approaches. The most straightforward one is the *Atomic Block* approach, which loads not only a matching prefix into the cache, but also finds all the sub-prefixes of the matching prefix in the FIB and loads them into the cache, therefore, subsequent packets will not encounter the cache-hiding problem. Another method *Uni-class* divides up a matching prefix



(a) Hit Ratios for One Router



(b) Hit Ratios for Different Approaches



(c) Hit Ratios With and Without Cache Initialization

Figure 4.10: Different Hit Ratios

into multiple fixed-length (24 bits) sub-prefixes on the fly and installs the one matching the destination address into the cache [46]. This approach assumes that 24 is the longest prefix length in the FIB, therefore, the cached /24 prefixes will not hide more specific prefixes in the FIB. This assumption is usually true as operators filter out prefixes longer than /24 to prevent route leaks. Figure 4.10(b) compares the hit ratios for the different caching approaches with a fixed cache size of 20K. Our approach has a 99.95% hit ratio on average. The Atomic Block approach has a 99.62% hit ratio and Uni-class approach has a 97.19% hit ratio, on average. Although the hit ratio of the Atomic Block approach is close to our approach, it takes much more time to maintain the cache as shown in Section 3.5. The difference

between the hit ratios of the Atomic Block approach and our scheme is due to the fact that the Atomic Block approach fills the cache with all the sub-prefixes of a matching prefixes, which may include many prefixes that will not match subsequent packets. On the other hand, our scheme creates only the most specific prefix that matches an arriving packet's destination address and thus, for a given cache size, our scheme covers more useful prefixes than the Atomic Block approach. The low hit ratio of the Uni-class approach is due to its fixed long prefix length (24). Given the same cache size, it can cover much fewer useful addresses than the other approaches.

Moreover, we compared our approach with three techniques proposed by Liu [50], CPTE, NPE and PPTE, using a static routing table (the author did not specify update handling algorithms). NPE does not increase the FIB size and has a 99.16% hit ratio on average. PPTE increases the FIB size by 13,384 and has a 99.48% hit ratio on average. CPTE expands the FIB trie into a complete binary tree and installs disjoint prefixes into the cache, thus it has the same hit ratio as our scheme (not shown in the figure), but it significantly increases the FIB size by more than two times from 371,734 to 1,131,635 prefixes. In our scheme, we only increase the full FIB size by 6,288 and reach a hit ratio of 99.95% on average. Finally, the RRC-ME algorithm proposed by Akhbarizadeh et al. [19] uses a binary tree (with no expansion) and only installs or generates a disjoint prefix into the cache on the fly, and it will have the same hit ratio as our scheme (not shown in the figure), but our update handling algorithm is much more efficient (Section 3.4).

3.3 Initial Traffic Handling

One of the biggest concerns for ISPs is how to handle the initial traffic when the cache starts with an empty set [83]. Instead of a cold start, we fill the initial empty cache completely with the shortest non-overlapping prefixes if there is no history of popular prefixes available. Figure 4.10(c) shows the initial traffic hit ratios. We used

the first 1 million packets to do the experiment. The top line represents the hit ratio with cache initialization and the lower line represents the one without cache initialization. After the first 100 packets, the initialized cache has a hit ratio of 85% and the un-initialized one has a hit ratio of only 65%. Their hit ratios are very close to each other once 100,000 packets are forwarded.

3.4 Routing Update Handling Performance

Figure 4.11 shows the routing update handling performance. The top curve represents the total number of RIB updates. The middle curve represents the total number of updates (8,348) pushed to cache including next-hop changes (8,251) and prefix deletions, while the bottom curve shows the number of prefix deletions (97), which is only 3.18% of the total number of RIB updates. Since very few updates are pushed to the cache, the updates have minimal influence on the cache hit ratio. On the other hand, in the RRC-ME approach [19], each updated prefix needs to be converted into two IP addresses first which are then looked up in the cache to discover the matching prefix. In the process, the cache will be interrupted twice if there is no matching prefix; otherwise, it gets three interruptions. Specifically, in the period of 24 hours, the previous work needs at least 523,754 (261,877 $\times$ 2) lookups of the cache as compared to our scheme that needs only 8,251 lookups.

3.5 Time Cost

Figure 4.12 compares the time cost to process all of the routing updates and data of the three approaches. We made two main observations: (a) the Atomic Block approach takes about four times longer to finish the same task than the other two approaches; (b) our approach takes almost the same time as the Uni-class approach, but our approach has a much higher hit ratio as shown previously.

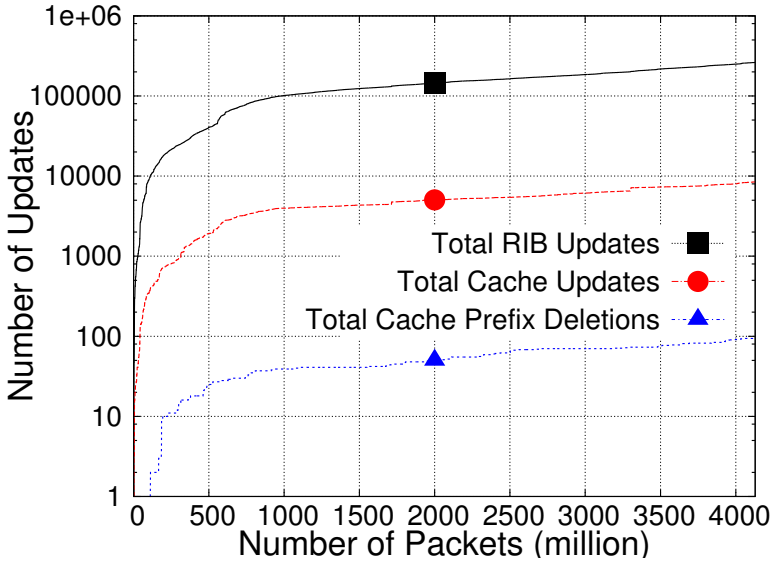


Figure 4.11: Update Handling Impact

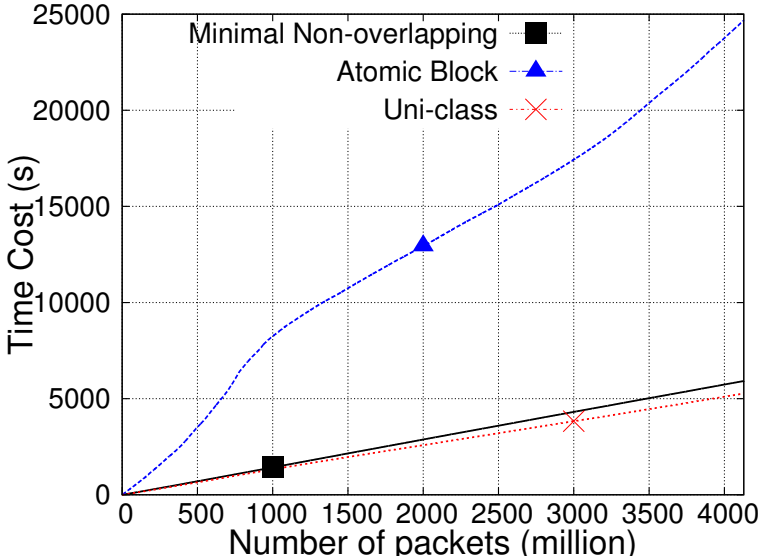


Figure 4.12: Total Time Cost

4 FIB Caching Implementation Using OpenFlow

Implementing FIB caching requires a way to define and control the forwarding table of a switch; OpenFlow provides an easy and powerful system for research and experimentation in programmable, logic-based switches. As many switches have similar methods for handling packets, the OpenFlow protocol allows us to program switches which implement the OpenFlow interface and make it possible to specify their forwarding table rules [56]. An OpenFlow switch maintains a flow table with corresponding actions for each flow entry and processes any incoming, and matching packets based on those actions. The OpenFlow switch communicates with an external controller that can install, remove, and modify flows from the switch's flow table. The external controller allows us to define desired behaviors for the switch without explicitly accessing or programming the switch.

The initial setup of our experiment's environment stays close to the OpenFlow Tutorial, which provides a step-by-step guide through the process of emulating an OpenFlow style network [10]. We created a virtual machine (VM) to host a simulation of a network topology and installed the VM image according to the tutorial, which provides a previously created virtual machine disc image with much of the software required for OpenFlow experimentation. The disc image included Mininet [8], the software used for creating and managing the testbed virtual network. Mininet is able to simulate multiple hosts and OpenFlow switches, as well as allow OpenFlow protocol communication with a remote controller. A packet inspector, Wireshark, and an OpenFlow packet dissector were also included with the provided disc image [17]. Wireshark gives us a way to view and analyze OpenFlow protocol messages for information and debugging purposes. Both our OpenFlow switch and our OpenFlow controller reside on the same server; the switch and hosts are simulated in Mininet, which runs on the virtual machine, and the controller runs

on the server natively.

The logic required for FIB caching must be implemented on the external controller; the controller handles any necessary decisions and modifies the switch's flow table accordingly. Our controller utilizes POX as its framework for interfacing with the OpenFlow switch; POX [12] is an SDN (Software Defined Networking) controller platform written in Python [12]. The FIB caching algorithm on which the controller relies is written in C and so the Python-based controller requires a method for interfacing with the C code. We use Boost.Python, a C++ library which gives Python the ability to interact with C++ classes and functions, to interface with our FIB caching code. By wrapping the necessary C code in a C++ interface, the controller may use the defined interface to access the caching algorithm through Python. The controller itself is programmed to use the FIB caching algorithm to determine which flows to install, remove, or modify on the OpenFlow switch.

The C++ interface provides the OpenFlow controller with five functions for interacting with the FIB caching C code. (1) Start Up: uses a text file containing the full RIB to build the FIB and RIB patricia tries. (2) Cache Initialization: uses the previously built FIB to create a list of prefixes that cover the most IP addresses; the length of the list is provided as an argument. (3) Get Initialized Cache: returns the initialized cache list. (4) Update: passes a route announcement to the FIB and performs the necessary operations. (5) Generate Prefix: performs a lookup in the FIB for the longest prefix which matches the packet provided as an argument. The OpenFlow controller handles communication with the OpenFlow switch and uses the C++ interface to make decisions on how it will interact with the flow table. When the controller first connects with the switch, the FIB and RIB patricia tries are built and an initialized cache is generated. After the initialized cache has been created, the controller installs a flow for each prefix; the flow defines its match rule so that any incoming packets which hit the initialized prefix are forwarded

according to that flow's actions. The controller also maintains an update file which contains the announcement and withdrawal of routes; the controller passes these updates to the interface for processing. On the event that a packet misses the switch's flow table, the switch notifies the controller by sending the packet which caused the miss. The controller uses the packet's network destination as an argument for the interface's generate prefix function and installs a flow using the prefix returned by the interface.

The controller maintains a count of the number of flows installed in the switch's flow table and also defines a limit to the number of active flows allowed. When a packet misses the switch's flow table and the flow table is full, the controller must choose the least useful flow to remove. Our controller is programmed with two different algorithms that it may utilize to perform a cache replacement: an LFU (Least Frequently Used) algorithm which inspects the packet count of the each flow in the switch's flow table and removes the flow with the lowest packet count and an LRU (Least Recently Used) algorithm which removes flows based on the time since a packet last matched the flow.

Our LFU algorithm depends on the statistics of the flows in the switch's flow table and so in the event of a full cache, the controller must request the full flow table from the switch. Once the controller has received the flow stats from the switch, it may request that the flow with the lowest packet count be removed from the flow table. Since multiple packets may miss and request flow stats from the switch before the controller has received a flow stat reply and chosen a suitable flow to remove, duplicate copies of the flow table may be returned to the controller. As the controller has no way to determine that two stat replies are the same, the controller will request the removal of the same flow twice. When the controller requests the removal of a flow from the switch's flow table, it must keep a list of flows queued for removal. The controller sorts the returned flows by packet count

and searches for the flow under consideration, which is the head of the sorted list initially, in the removal queue. If the flow under consideration is not found in the removal queue, the controller requests for the flow to be removed and the flow is added to the removal queue. If the flow under consideration is found to be in the removal queue, the next flow in the sorted list is considered until a suitable flow is found. When the flow is finally removed from the flow table, the switch sends a notification informing the controller the flow has been removed, and the controller may then remove the flow from the removal queue.

Our LRU algorithm depends on the idle time of each flow in the flow table and thus is not handled directly by the controller. The controller installs flows with a maximum idle time, the time a flow can be idle before it is removed from the flow table, and the switch handles and removes any flows that exceed their idle time. Since the controller is keeping a count of the number of flows in the flow table, it can adjust the idle time before installing new flows and modify the idle time of currently installed flows based on the installed flow count. As OpenFlow does not provide a means besides idle flow timeout for cache replacement, the implementation of our desired algorithms was difficult. While a switch does maintain the time a flow has been idle, the controller does not have access to this information.

An LRU algorithm where the controller removes a single flow that has been used least recently is therefore impossible. The controller is also unable to request statistics or a single flow based on a specified flow variable. Our LFU algorithm must request the entire flow table and sort the returned flows to choose a single flow with the lowest packet count. It is also necessary for the controller to continue installing flows even when the flow table is full because of the delay between a statistics request and a statistics reply. In the case of a switch with 20K flows installed in the flow table, a reply from the switch takes around five seconds; an impractical time to wait because packets that have missed the flow table are still

being sent to the controller to be processed. The usefulness of OpenFlow for research and experimentation would be greatly increased with the inclusion of switch-defined or controller-accessible cache replacement methods. We are doing further experiments now.

Chapter 5

Conclusion

FIB aggregation is a promising direction in controlling the growing FIB size. We have proposed FIFA, a suite of three FIB aggregation algorithms with significant performance improvement over existing algorithms in terms of time cost, total number of FIB changes and FIB burst size. For our next step, we plan to investigate how to automatically switch among the algorithms based on a set of constraints, e.g., memory size, aggregated FIB size and maximum FIB burst size.

We presented an effective caching scheme to mitigate the problems caused by the rapidly increasing size of the global forwarding table. This scheme allows ISPs to keep their operational cost low by storing a fraction of the full FIB in the expensive fast memory, while storing the full FIB in slow memory. Our results based on real data show that we can use only 3.5K prefixes to reach a hit ratio of 99.25% and 20K prefixes to reach a hit ratio of 99.95%. Moreover, we fill the initial empty cache with the shortest non-overlapping prefixes and obtain a high hit ratio for the initial traffic. Finally, our scheme includes a set of efficient algorithms to process both FIB and cache update events, while preventing the cache-hiding problem. These results are specifically for a regional network, we plan to try more datasets to justify our conclusion.

Appendix A

Proofs of Algorithms

1 Correctness Proof of Improved ORTC Implementation

1.1 Definitions

ORTC algorithm uses a binary tree B to implement the aggregation algorithm and FIFA-S uses a Paricia Trie T to improve ORTC. Each node x (or prefix) on the binary tree or Patricia Trie contains the following fields for aggregation computation.

$O_t(x)$: the original next-hop of x for a trie or tree t.

$M_t(x)$: the next-hop set of x for a trie or tree t.

$S_t(x)$: the selected next-hop of x for a trie or tree t.

$Len(x)$: the length of prefix x for a trie or tree t.

In ORTC, an operation was defined to obtain merged next-hops: $X\#Y$, if the next-hop set of X has a common element(s) with the next-hop set of Y, then do intersection operation; otherwise, do union operation.

At beginning, for the binary tree B and Patricia Trie T, $O_B(x) = O_T(x)$: B and T have the same original next-hops for each prefix x. Also they use two passes to finish the aggregation: merging next-hops and selecting next-hops(including generating new nodes)

First, we define merging next-hop operations as follows:

For both B and T, $M(x) = O(x)$ for leaf nodes

For the Patricia Trie T, the operations on other nodes are defined as follows: $M_T(x) = M_T\_l(x)$ # $M_T\_r(x)$, $M_T\_l(x)$ and $M_T\_r(x)$ are not the directly connected left and right child next-hop sets, but are imaginary sets. The method to obtain $M_T\_l(x)$ and $M_T\_r(x)$ are as follows:

$M_T\_l(x)$:

- $O_T(x)$, if x has no left child.
- $M_T(y)$, if $Len(y) = Len(x) + 1$, y is the prefix of the left child.
- $M_T(y)$ # $O_T(x)$, if $Len(y) = Len(x) + 2$.
- $O_T(y)$, if $Len(y) > Len(x) + 2$.

$M_T\_r(x)$:

- $O_T(x)$, if x has no left child.
- $M_T(y)$, if $Len(y) = Len(x) + 1$, y is the prefix of the right child.
- $M_T(y)$ # $O_T(x)$, if $Len(y) = Len(x) + 2$.
- $O_T(x)$, if $Len(y) > Len(x) + 2$.

For the binary tree B, the operations on other nodes are defined as follows: $M_B(x) = M_B\_l(x)$ # $M_B\_r(x)$, here M_l and M_r are the directly connected left and right child next-hop sets for B. $M_B\_l(x)$=$M_B(y)$ and $M_B\_r(x)$=$M_B(y)$, since $Len(y) = Len(x) + 1$ and x is not a leaf node.

Second, we have the generating missing nodes and selecting next-hops process for x (let w be INFIB ancestor of x) on T:

- If x is the root. In this case, if $O(x)$ in $M(x)$, then $S(x) = O(x)$; otherwise, the first next-hop from $M(x)$ will be picked as the selected next-hop.
- Otherwise (x is not the root)
  - If $S(w)$ is in $M(x)$, then $S(x) = S(w)$ and $F(x) = 0$; otherwise, $S(x) = M(x)[0]$ and $F(x) = 1$.
  - If $S(x) \neq O(x)$, then we need to generate a missing node. If the left or right child is missing for x, then we generate a child y, where $Len(y) = Len(x)+1$, and set $O(y)=O(x)$, $M(y)=O(y)$. if y exists and is a child of x, in this case, two possible cases may happen. First If $Len(y) > Len(x) + 1$, then we generate a child node z, where $Len(z) = Len(x)+1$, $O(z)=O(x)$, $M(z)=O(z)$; second, if $Len(y) > Len(x) + 2$, then we generate a node with $M(z)=M(y)\#O(z)$.
- We run the same procedure $P(x, w)$ for all children of x.

We define the next-hop selection process for x (let w be INFIB ancestor of x) on B:

- If x is the root, if $O(x)$ is in $M(x)$, then $S(x) = O(x)$. Otherwise, the first one from $M(x)$ is picked as the selected next-hop.
- Otherwise (x is not the root). If $S(w)$ is in $M(x)$, then $S(x) = S(w)$ and $F(x) = 0$; otherwise, $S(x) = M(x)[0]$ and $F(x) = 1$.
- We run the same procedure $P(x, w)$ for all children of x.

1.2 Theorem 1.1

Theorem 1.1. *The aggregation operations (bottom up merging next-hops and top down selecting next-hop) on a T will obtain the same results as the ones from a B using ORTC (two passes).*

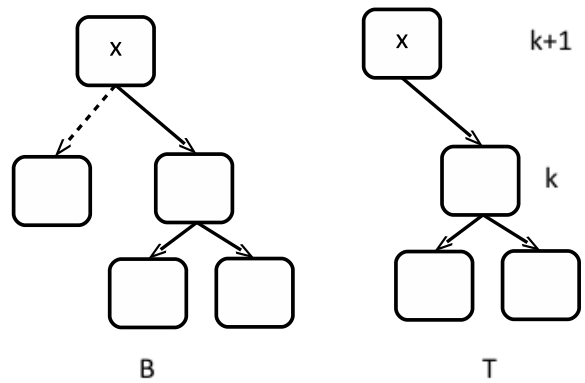


Figure A.1: Lemma 1.2 Case a

1.3 Lemma 1.2

Lemma 1.2. *For every node in a T with a prefix x must have the same merged next-hop set as the one in a B with the same prefix, namely, $M_T(x) = M_B(x)$ for every prefix x on T.*

proof:

We use induction to prove the lemma, first we assume bottom up from a T, the level starts from 1 to n. Notice that the prefix length difference between two levels may be greater than 1.

When $n = 1$, $M(x) = O(x)$ in both Patricia Trie T and Binary Tree B, thus the statement is true.

Assume $n = k$, the statement is true, namely, all y at level k, $M_T(y) = M_B(y)$. Now we need to prove when $n = k + 1$, the statement is still true, $M_T(x) = M_B(x)$ where x is the parent node of y.

There are four cases to consider as follows:

- Case a (one child is missing): we know $M_B\_l(x) = O_B(x)$, $M_T\_l(x) = O_T(x)$ and $O_B(x) = O_T(x)$ by definition, thus $M_B\_l(x) = M_T\_l(x)$. See Figure A.1.

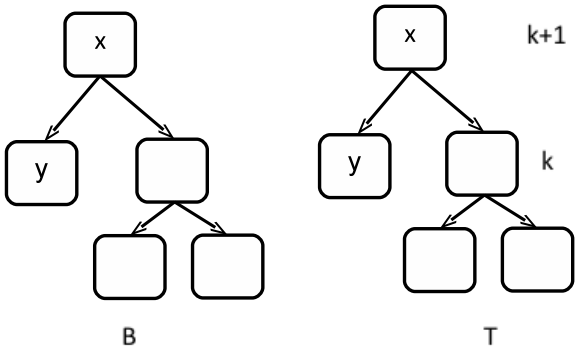


Figure A.2: Lemma 1.2 Case b

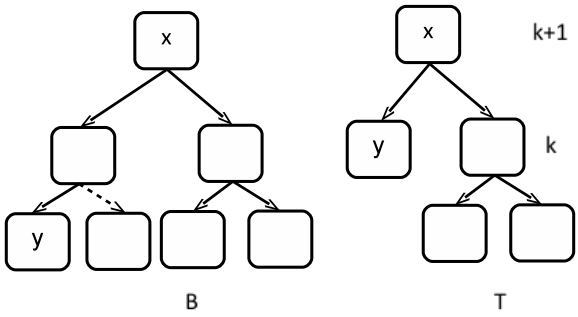


Figure A.3: Lemma 1.2 Case c

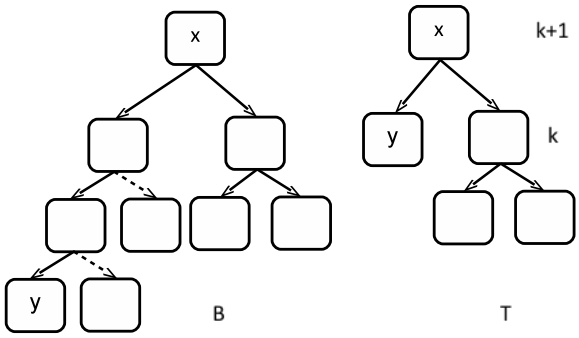


Figure A.4: Lemma 1.2 Case d

- Case b ($Len(y) - Len(x)$=1): y is at level k, we know $M_B(y)$=$M_T(y)$ by assumption, $M_B\_l(x) = M_B(y)$ and $M_T\_l(x) = M_T(y)$ by definition, thus $M_T\_l(x) = M_B\_l(x)$. See Figure A.2.

- Case c ($Len(y) = Len(x)$+2): we know $M_B\_l(x) = M_B(y)\#O_B(x)$ by definition and leaf pushing, $M_T\_l(x) = M_T(y)\#O_T(x)$ by definition and $M_T(y) = M_B(y)$ by assumption, thus $M_T\_l(x) = M_B\_l(x)$. See Figure A.3.

- Case d ($Len(y) > Len(x)$+2): we know $M_B\_l(x) = M_B(y)\#O_B(x)\ldots\#O_B(x) = O_B(x)$ by leaf pushing definition, $M_T\_l(x, T) = O_T(x)$ and $O_T(x) = O_B(x)$ by definition, thus $M_T\_l(x, T) = M_B\_l(x)$. See Figure A.4.

We proved for all cases $M_T\_l(x, T) = M_B\_l(x)$ and the same for $M_T\_r(x) = M_B\_r(x)$, therefore, $M_T(x) = M_B(x)$, we proved *Lemma* 1.2.

1.4 Lemma 1.3

Lemma 1.3. *The next-hop selection process of both T and B should output the same prefixes and next-hops into FIB, namely, $S_T(x) = S_B(x)$ and $F_T(x) = F_B(x)$ for every prefix x on T.*

Proof:

we use induction approach to top down prove the statement from level 1 to level n.

First, when $n = 1$, in *Lemma* 1.2, we proved $M_T(x) = M_B(x)$ for every node in T, and we apply the same selection rules P on both sets and the missing node generation rules on T, then $S_T(x) = S_B(x)$. Now assume when $n = k$, *Lemma* 1.3 is true, we need to show when $n = k$+1, *Lemma* 1.3 is still true. We also assume if a new node needs to be generated at a new level over level k, then the new node is at level k+1.

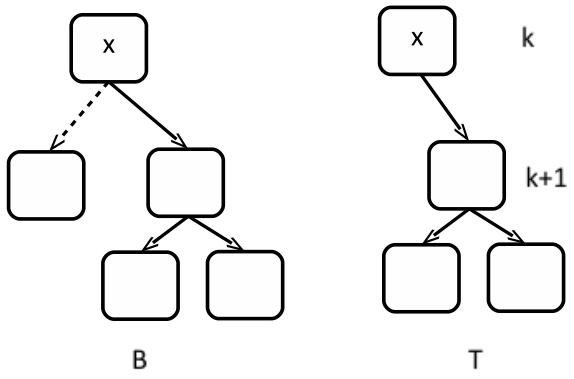


Figure A.5: Lemma 1.3 Case 1a

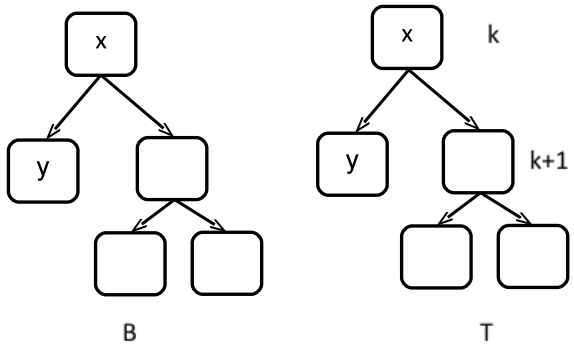


Figure A.6: Lemma 1.3 Case 1b

In the improved ORTC algorithms, we use the $INFIB$ ancestor w to pick selected next-hop. Actually we could use the selected next-hop of the directly connected parent x, because $S_T(x) = S_T(w)$ is always true, where w is the INFIB ancestor of x. Here is the proof: if $S_T(x)$ is in FIB, then $x = w$; If $S_T(x)$ is not in FIB, they must have the same selected next-hops, then $S_T(x) = S_T(w)$.

- Case 1: $S(x) = O(x)$
  - Subcase a (no left or right child): we know $S_B(x) = S_T(x)$ by assumption and $M_B(z) = O_B(x)$ due to leaf pushing; also we know $S_B(z)$ is in $M_B(z)$ because z is a leaf node and $M_B(z) = O_B(z) = O_B(x) = S_B(x)$, thus

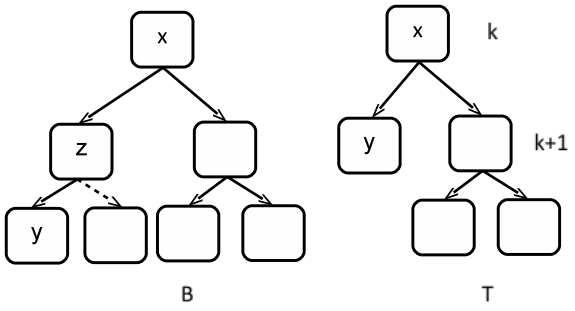


Figure A.7: Lemma 1.3 Case 1c

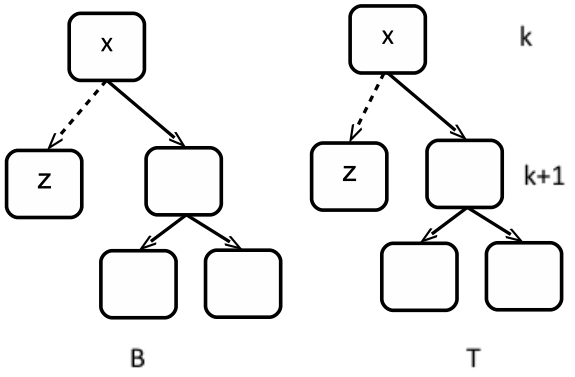


Figure A.8: Lemma 1.3 Case 2a

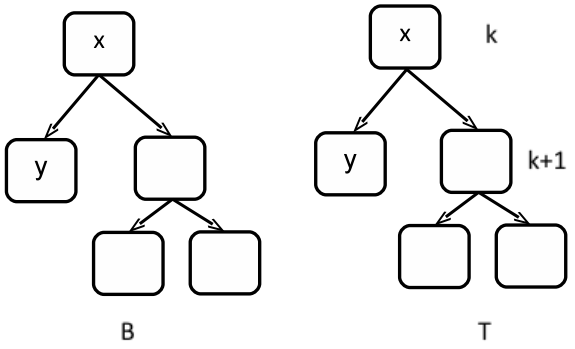


Figure A.9: Lemma 1.3 Case 2b

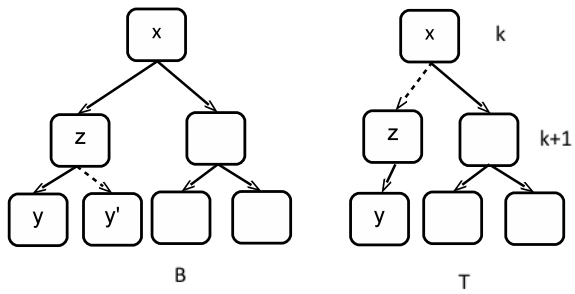


Figure A.10: Lemma 1.3 Case 2c

$S_B(z) = S_B(x)$ and $F_B(z) = 0$ by following the next-hop selection rules. In this case, we do nothing for T and $F_T(z) = 0$, thus we see that neither prefix from B and T will be put into FIB. See Figure A.5.

– Subcase b ($len(y) = len(x)+1$): we know $M_B(y) = M_T(y)$ by the proof of *Lemma* 1.2; also we know $S_B(x) = S_T(x)$ by assumption and $O_B(y) = O_T(y)$ by definition. If $S_B(x)$ is in $M_B(y)$, then $S_B(y) = S_B(x)$ and $F_B(y) = 0$. In this case, $S_T(x)$ is also in $M_T(y)$, then $S_T(y) = S_T(x) = S_B(x) = S_B(y)$ and $F_T(y) = 0$. Otherwise, $S_B(y) = M_B(y)$ [0], $F_B(y) = 1$. In this case, $S_T(x)$ is also not in $M_T(y)$, then $S_T(y) = M_T(y)$ [0] $= M_B(y)$ [0] $= S_B(y)$ and $F_T(y) = 1$. See Figure A.6.

– Subcase c ($len(y) > len(x)+1$): We know $S_B(x) = S_T(x)$ by assumption; $M_B(z)$ contains $O_B(x)$ by leaf pushing; $S_B(x) = O_B(x)$ and $O_B(x)$ is in $M_B(z)$, thus we can obtain that $S_B(z) = S_B(x)$ by the next-hop selection rules and $F_B(z) = 0$; $M_B(y) = M_T(y)$ from *Lemma* 1.2; If $S_B(z)$ is in $M_B(y)$, then $S_B(y) = S_B(z) = S_B(x)$, $F_B(y) = 0$. In this case, $M_T(y) = M_B(y)$ by *Lemma* 1.2, $S_T(x) = S_B(x) = S_B(z)$, thus $S_T(x)$ in $M_T(y)$ and then $S_T(y) = S_T(x)$, and $F_T(y) = 0 = F_B(y)$. If $S_B(z)$ is not in $M_B(y)$, then $S_B(y) = M_B(y)$ [0], $F_B(y) = 1$; In this case, $S_T(x) = S_B(x) = S_B(z)$ and $M_B(y) = M_T(y)$, thus $S_T(x)$ is not in $M_T(y)$, thus $S_T(y) =$

$M_T(y)$ [0] $= M_B(y)$ [0] $= S_B(y)$ and $F_T(y) = 1 = F_B(y)$ by following the next-hop selection rules; thus $S_B(y) = S_T(y)$ and $F_B(y) = F_T(y)$. See Figure A.7.

- Case 2: $S(x) \neq O(x)$
  - Subcase a (Missing one child): We know $S_B(x) = S_T(x)$ by induction, in B, z is a leaf pushed node with $M_B(z) = O_B(x)$; since $S_B(x) \neq O_B(x)$, by following the next-hop selection rules $S_B(y) = M_B(x)$ [0] and $F_B(y) = 1$; From *Lemma* 1.2, we know $M_T(z) = M_B(z)$, thus by following the node generating rules, y will be generated in T with $O_T(z) = O_T(x)$ and $M_T(z) = O_T(x)$; $S_T(x) \neq O_T(x)$, thus $S_T(z) = M_T(z)$ [0] $= M_B(z)$ [0] $= S_B(z)$ and $F_T(z) = 1 = F_B(z)$ by following the next-hop selection rules, accordingly $S_B(y) = S_T(y)$ and $F_B(y) = F_T(y)$. See Figure A.8.
  - Subcase b ($Len(y) = Len(x)+1$): By *Lemma* 1, we know $M_B(y) = M_T(y)$ and $S_B(x) = S_T(x)$ by assumption; Also we defined $O_B(y) = O_T(y)$. If $S_B(x)$ is in $M_B(y)$, then $S_B(y) = S_B(x)$ and $F_B(y) = 0$. In this case, $S_T(x)$ is also in $M_T(y)$, then $S_T(y) = S_T(x) = S_B(x) = S_B(y)$ and $F_T(y) = 0$. Otherwise, in B, $S_B(y) = M_B(y)$ [0], $F_B(y) = 1$. In this case, $S_T(x)$ is also not in $M_T(y)$, then $S_T(y) = M_T(y)$ [0] $= M_B(y)$ [0] $= S_B(y)$ and $F_T(y) = 1$. See Figure A.9.
  - Subcase c ($Len(y) > Len(x)+1$): By induction, $S_B(x) = S_T(x)$ and z will be generated in T by the definition of generating node rules; also we defined $M_B(z) = M_B(y)$ $\#O_B(x)$ if $len(y) = len(x)+2$, $M_B(z) = M_B(y)$ $\#O_B(x)$ $\#O_B(x) = O_B(x)$ if $len(y) > len(x)+2$, $M_T(z) = M_B(y)$ $\#O_T(x)$ if $len(y) = len(x) + 2$ and $M_T(z) = O_B(x)$ if $len(y) > len(x) + 2$; thus we can prove that $M_B(z) = M_T(z)$. If $S_B(x)$ is in $M_B(z)$ then $S_B(z) = S_B(x)$ and $F_B(z) = 0$; in this case, $S_B(x) = S_T(x)$ and $O_B(z) =$

$O_T(z)$ and $M_B(z)$ =$M_T(z)$, then $S_T(z) = S_T(x) = S_B(x) = S_B(z)$ and $F_T(z) = 0 = F_B(z)$; otherwise, if $S_B(x)$ is not in $M_B(z)$, then $S_B(z) = M_B(z)$ [0] and $F_B(z) = 1$; in this case, $S_B(x) = S_T(x)$ and $O_B(z) = O_T(z)$ and $M_B(z)$ =$M_T(z)$, then $S_T(z) = M_T(z)$ [0]=$M_B(z)$ [0] = $S_B(z)$ and $F_T(z) = 1 = F_B(z)$. Therefore, $S_B(z)$ =$S_T(z)$ and $F_B(z) = F_T(z)$. See Figure A.10.

Therefore, we proved *Lemma* 1.3 for all cases. *Theorem* 1.1 consists of *Lemma* 1.2 and *Lemma* 1.3, therefore, we proved *Theorem* 1.1 as well.

2 FIFA-S proofs

2.1 Lemma 2.1

Lemma 2.1. *If an update does not affect the original next-hops of every node on a subtree, the merged next-hop set of every node on the subtree will not change after applying the first step of the patricia trie based re-aggregation.*

Proof

Assume before update, next-hops are O^{old}, M^{old}, M^{old}_l, M^{old}_r, S^{old} for an original next-hop, merged next-hop, imaginary left merged next-hop, imaginary right merged next-hop and selected next-hop. Before and after re-aggregation, two tries have the same old next-hops. After update, they are O^{new} , M^{new} , M^{new}_l , M^{new}_r , S^{new} for the same set of next-hops. For T, the merging next-hop process will follow normal patricia trie merging next-hop process. In the subtrees with unchanged original next-hops, the deepest level is at level 1 for T.

When level $n = 1$, all nodes are leaf nodes, $O_T^{new}(x) = O_T^{old}(x)$ and $M_T^{new}(x) = O_T^{new}(x) = O_T^{old}(x) = M_T^{old}(x)$.

We assume when $n = k$, $M_T^{new}(y) = M_T^{old}(y)$, we need to prove that when $n = k + 1$, $M_T^{new}(x) = M_T^{old}(x)$. There are four cases to cover as follows:

- Case a (One left child is missing): we defined $M_T^{new}\_l(x) = O_T^{new}(x)$ and we assume $O_T^{new}(x) = O_T^{old}(x)$ in the induction; also we know $M_T^{old}\_l(x) = O_T^{old}(x)$ by definition; thus $M_T^{new}\_l(x) = M_T^{old}\_l(x)$.

- Case b ($Len(y) = Len(x) + 1$): we assume $M_T^{new}(y) = M_T^{old}(y)$ by induction and defined $M_T^{new}\_l(x) = M_T^{new}(y)$ and $M_T^{old}\_l(x) = M_T^{old}(y)$; thus we can obtain $M_T^{new}\_l(x) = M_T^{new}(y) = M_T^{old}(y) = M_T^{old}\_l(x)$.

- Case c ($Len(y) = Len(x) + 2$): we can know $M_T^{new}\_l(x) = M_T^{new}(y) \# O_T^{new}(x)$ by *Lemma* 1.2 and $M_T^{new}(y) = M_T^{old}(y)$ by induction; also we know $M_T^{old}\_l(x) = M_T^{old}(y) \# O_T^{old}(x)$ by by *Lemma* 1.2 and $O_T^{new}(x) = O_T^{old}(x)$ by assumption; as a result, $M_T^{new}\_l(x) = M_T^{new}(y) \# O_T^{new}(x) = M_T^{old}(y) \# O_T^{old}(x) = M_T^{old}\_l(x)$.

- Case d ($Len(y) > Len(x) + 2$): we know $M_T^{new}\_l(x) = M_T^{new}(y) \# O_T^{new}(x) \# \# O_T^{new}(x) = O_T^{new}(x)$ by *Lemma* 1.2 and $O_T^{old}(x) = O_T^{new}(x)$ by definition; also we defined $M_T^{old}\_l(x) = M_T^{old}(y) \# O_T^{old}(x) \# \ldots \# O_T^{old}(x) = O_T^{old}(x)$; as a result, $M_T^{new}\_l(x) = O_T^{old}(x) = O_T^{new}(x) = M_T^{new}\_l(x)$.

In all of the above cases, $M_T^{new}\_r(x) = M_T^{old}\_r(x)$, thus $M_T^{new}(x) = M_T^{new}\_l(x) \# M_T^{new}\_r(x) = M_T^{old}\_l(x) \# M_T^{old}\_r(x) = M_T^{old}(x)$. Therefore, we proved *Lemma.* 2.1.

2.2 Lemma 2.2

Lemma 2.2. *If an update does not affect the original next-hops of every node on a subtree and the second step of the patricia trie based re-aggregation does not change the selected next-hop of the subtree root, then the selected next-hop sets and FIB*

status of other nodes on the subtree will not change after applying the second step of the patricia trie based re-aggregation.

Proof

We employ an induction approach to show the proof. We know the input is: $S^{new}(x) = M^{old}(x)$ for the subtree root, $O^{new}(x) = O^{old}(x)$ for all subtree nodes and $M^{new}(x) = M^{old}(x)$ by *Lemma* 2.1. We need to top down prove the lemma from level 1 to level n.

First, when n = 1, in *Lemma* 2.1, we know $M_T^{new}(x) = M_T^{old}(x)$ for every node in T, and we apply the same selection rules P on $M_T^{new}(x)$ and $M_T^{old}(x)$ as well as the same rules to create the missing nodes on T, then $S_T^{new}(x) = S_T^{old}(x)$ and $F_T^{new}(x)$ $F_T^{old}(x)$.

Now assume when $n = k$, *Lemma* 2.2 is true, we need to show when $n = k + 1$, *Lemma* 2.2 is still true. We also assume if a new node needs to be generated at a new level over level k, then the new node is at level $k + 1$.

In the FIFA algorithms, we use the $INFIB$ ancestor w to pick selected next-hop. Actually we could use the selected next-hop of the directly connected parent x, because $S_T(x) = S_T(w)$ is always true, where w is the INFIB ancestor of x, which has been proved in *Lemma* 1.3.

- Case 1 ($S^{new}(x) = O^{new}(x)$):
  - Subcase a (No left or right child): we know $S_T^{new}(x) = S_T^{old}(x)$ by assumption. we need to do nothing for the missing child for re-aggregation in this case.
  - Subcase b ($len(y) >= len(x)$+1): we know $M_T^{new}(y) = M_T^{old}(y)$ by *Lemma* 2.1 and $S_T^{new}(x) = S_T^{old}(x)$ and $F_T^{new}(x) = F_T^{old}(x)$ by assumption, also $O_T^{old}(y) = O_T^{old}(y)$ by definition. If $S_T^{new}(x)$ is in $M_T^{new}(y)$, then $S_T^{new}(y) = S_T^{new}(x)$ and $F_T^{new}(y) = 0$. In this case, we could infer $S_T^{old}(x) = S_T^{new}(x)$

by assumption and $M_T^{new}(y) = M_T^{old}(y)$, thus $S_T^{new}(y) = S_T^{old}(y) = S_T^{old}(x)$ and $F_T^{new}(y) = 0$. Otherwise, $S_T^{new}(y) = M_T^{new}(y)$ [0], $F_T^{new}(y) = 1$. In this case, we could infer $S_T^{new}(x) = S_T^{old}(x)$ and $M_T^{new}(y) = M_T^{old}(y)$, then $S_T^{new}(y) = M_T^{new}(y)$ [0] $= M_T^{old}(y)$ [0] $= S_T^{old}(y)$ and $F_T^{new}(y) = 1$.

- Case 2 ($S^{new}(x) \neq O^{new}(x)$):
  - Subcase a (Missing one child): we defined $S^{new}(x) = M^{old}(x)$ and $O^{new}(x) = O^{old}(x)$. Since $M^{old}(x) \neq O^{old}(x)$, then previous aggregation will create a new node for this case. This will not happen during an update.
  - Subcase b (len(y)=len(x)+1): We know $M_T^{new}(y) = M_T^{old}(y)$ by *Lemma* 2.1 and$S_T^{new}(x) =S_T^{old}(x)$ and $F_T^{new}(x) = F_T^{old}(x)$ by assumption; we also defined $O_T^{new}(y) = O_T^{old}(y)$. If $S_T^{new}(x)$ is in $M_T^{new}(y)$, then $S_T^{new}(y) = S_T^{new}(x)$ and $F_T^{new}(y) = 0$. In this case, $S_T^{old}(x) = S_T^{new}(x)$ by assumption and $M_T^{new}(y) = M_T^{old}(y)$, then $S_T^{new}(y) = S_T^{old}(y) = S_T^{old}(x)$ and $F_T^{new}(y) = 0$. Otherwise, $S_T^{new}(y) = M_T^{new}(y)$ [0], $F_T^{new}(y) = 1$. In this case, $S_T^{new}(x) = S_T^{old}(x)$ and $M_T^{new}(y) = M_T^{old}(y)$, then $S_T^{new}(y) = M_T^{new}(y)$ [0] $= M_T^{old}(y)$ [0] $= S_T^{old}(y)$ and $F_T^{new}(y) = 1$.
  - Subcase c ($len(y) > len(x)$+1): This will not happen because the previous aggregation creates a new node between x and y the same as Subcase a.

Therefore, for all cases, *Lemma* 2.2 is true and we proved it.

2.3 Theorem 2.3

Theorem 2.3. *FIFA-S yields optimal aggregation for each update, with the same results as in re-aggregation from scratch.*

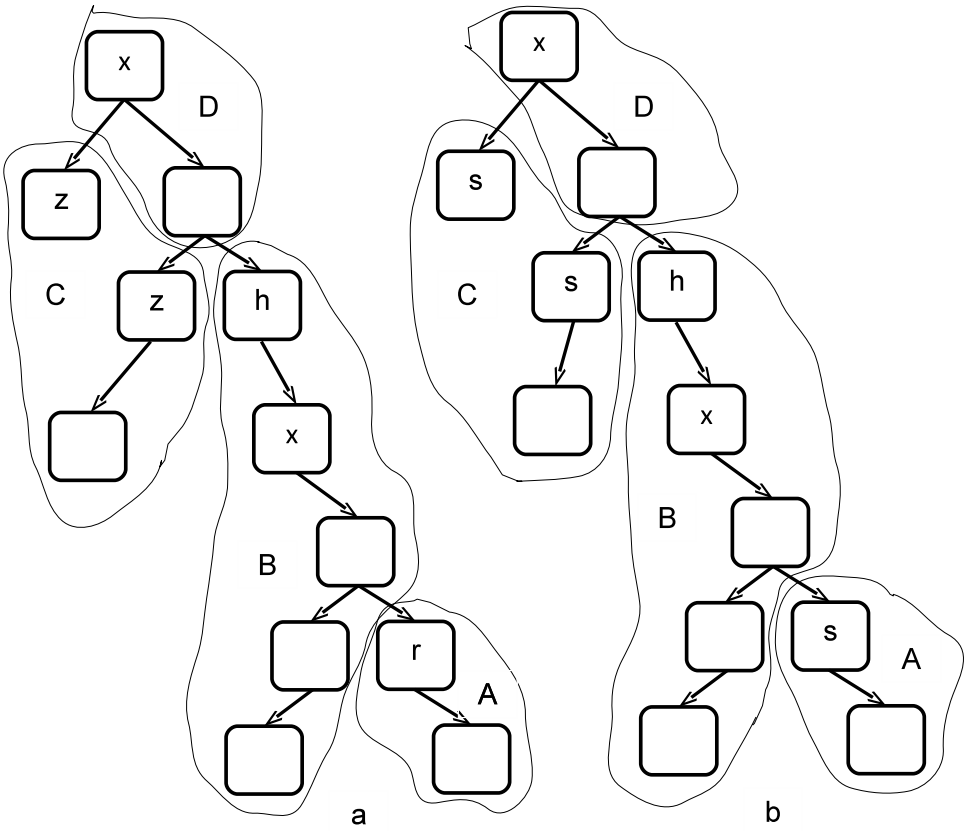


Figure A.11: Theorem 2.3

2.4 Lemma 2.4

Lemma 2.4. *The bottom-up next-hop merging process of FIFA-S will obtain the same merged next-hops as re-aggregation*

FIFA-S Merge Next Hop Definition

- Assume the highest changed node prefix h and the update prefix is u.
- If x is in the subtree rooted at a REAL node which is a prefix below u, no need for merging on the subtree rooted at r (Area A).
- If x is a prefix between h and root R, no need to conduct merging (Area D).
- If x is on the subtree rooted at z, then no need to conduct merging (Area C).
- Otherwise, $M^{new}(x) = M^{new}\_l(x) \# M^{new}\_r(x)$, same operations as re-aggregation (Area B).

Proof

- h is the the highest changed node for merged next-hop set. In the bottom up merging next-hop process, compare the old merged next-hop set and the newly generated one, h is the last one with $M^{new} \neq M^{old}$; r is a REAL prefix under update prefix u; z represents any prefix whose parent prefix is one node between u and root (including root but not u).
- The whole tree can be partitioned into four areas. In area A and C, their original next-hop do not change, thus $M^{new} = M^{old}$, therefore, re-aggregation will lead to the same results as the FIFA-S without the merging process. In area B, re-aggregation and FIFA-S have the same operations, thus their results must be the same. In area D, re-aggregation will obtain the same results as the one without any merging due to the highest change node definition.

Therefore, *Lemma* 2.4 has been proved.

2.5 Lemma 2.5

Lemma 2.5. *The top-down next-hop selection process of FIFA-S will obtain the same selected next-hops as re-aggregation*

- Assume the highest changed node prefix h and the update prefix is u; s is a prefix under update prefix u satisfying conditions of *Lemma* 4 (selected next-hop of the prefix does not change and all original and merged next-hops of all subtree nodes do not change); z represents any prefix satisfying conditions of *Lemma* 4.
- If x is in one subtree where it satisfies conditions of *Lemma* 4, no need to do next-hop selection on the subtree (Area A and C).
- If x is a prefix between h and the root, then no need to do next-hop selection (Area D).
- Otherwise, from h, follow the same rules of re-aggregation(Area B).

Proof

- In area D, since the merged next-hop set did not change before and after bottom up merging next-hop process, the selected next-hops will not change either, because they follow the same next-hop selection rules, therefore, no need to perform re-aggregation operations in this area.
- In area A and C, by *Lemma* 4, the original next-hops do not change on the subtree, thus no need to perform re-aggregation operations in these areas.
- In area B, re-aggregation and FIFA-S have the identical operations, therefore, they will produce the same results.

- Area A, B, C and D cover the entire tree, therefore, FIFA-S produces the same optimal results as re-aggregation.

Therefore, *Lemma* 2.5 has been proved.

Theorem 2.3 has two steps which are *Lemma* 2.4 and *Lemma* 2.5, therefore, it has been proved.

3 FIFA-T and FIFA-H proofs

3.1 FIFA-T proof

- FIFA-T includes bottom up merging next-hops to the update prefix and top down selecting next-hop from the update prefix.

- The process above is exactly the same as improved ORTC over the subtree rooted at the update prefix, thus the forwarding correctness can be guaranteed. Once a threshold is reached, do re-aggregation and it will obtain optimal results, the forwarding correctness can be guaranteed as well.

Therefore, FIFA-T has been proved.

3.2 FIFA-H proof

- Before a threshold is reached, FIFA-H includes bottom up to merging next-hops to a CAP prefix (CAP prefix is always higher than or equal to the update prefix) and top down to selecting next-hops from the update prefix. The above process will obtain the same results as improved ORTC over the subtree rooted at the update prefix and thus the forwarding correctness can be guaranteed.

- After a threshold is reached, it includes bottom up merging next-hops to a CAP level and top down selecting next-hops from the CAP prefix. The above process will obtain the same results as improved ORTC over the subtree rooted at the CAP prefix and thus the forwarding correctness can be guaranteed.

Therefore, FIFA-H has been proved.

Bibliography

[1] Bgpimple-simple BGP peering and route injection script.
http://code.google.com/p/bgpsimple/wiki/README.

[2] Cisco router.
http://www.cisco.com/en/US/products/hw/routers/index.html.

[3] Cisco Router Memories. http://archive.networknewz.com/networknewz-10-20050125MemoriesofaCiscoRouter.html.

[4] FIFA presentation. https://www.nanog.org/meetings/nanog56/abstracts.php?pt=MjAxNCZuYW5vZzU2&nm=nanog56.

[5] IETF Global Routing Operations (GROW).
http://www.ietf.org/dyn/wg/charter/grow-charter.html.

[6] IRTF Routing Research Group.
http://www.irtf.org/charter?gtype=rg&group=rrg.

[7] Juniper router.
http://www.juniper.net/us/en/products-services/routing/.

[8] Mininet. http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet.

[9] Net-Patricia Perl Module. http://search.cpan.org/dist/Net-Patricia/.

[10] OpenFlow Tutorial.
http://www.openflow.org/wk/index.php/OpenFlow_Tutorial.

[11] Opportunistic Topological Aggregation in the RIB-FIB Calculation? http://www.ops.ietf.org/lists/rrg/2008/threads.html#01880.

[12] Python. http://www.boost.org/doc/libs/1_53_0/libs/python/doc.

[13] Quagga. http://www.nongnu.org/quagga/.

[14] Router. https://en.wikipedia.org/wiki/Router_%28computing%29.

[15] Router Memories. http://community.brocade.com/community/blogs/service_providers/blog/2012/03/09/a-perspective-on-router-architecture-challenges-part-1-memory.

[16] Tabledump. https://bitbucket.org/ripencc/bgpdump/wiki/Home.

[17] Whiteshark. http://www.wireshark.org/.

[18] Advanced Network Technology Center at University of Oregon. The RouteViews project. http://www.routeviews.org/.

[19] M. J. Akhbarizadeh and M. Nourani. Efficient prefix cache for network processors. In *Proceedings of the High Performance Interconnects*, 2004.

[20] R. Atkinson and S. Bhatti. ILNP Architectural Description. Work in Progress, http://tools.ietf.org/html/draft-irtf-rrg-ilnp-arch-06.

[21] H. Ballani, P. Francis, C. Tuan, and J. Wang. Making Routers Last Longer with ViAggre. In *NSDI*, 2009.

[22] M. Bando, Y.-L. Lin, and H. J. Chao. Flashtrie: beyond 100-gb/s ip route lookup using hash-based prefix-compressed trie. *Networking, IEEE/ACM Transactions on*, 20(4):1262–1275, 2012.

[23] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)*, 24(3):293–318, 1992.

[24] T. Bu, L. Gao, and D. Towsley. On Characterizing BGP Routing Table Growth. *Computer Networks*, 45(1):45–54, may 2004.

[25] G. Cheung and S. McCanne. Optimal routing table design for ip address lookups under memory constraints. In *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1437–1444. IEEE, 1999.

[26] J. Cocke. Global common subexpression elimination. *SIGPLAN Not.*, 5(7):20–24, July 1970.

[27] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. *Small forwarding tables for fast routing lookups*, volume 27. ACM, 1997.

[28] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small forwarding tables for fast routing lookups. In *in ACM Sigcomm*, pages 3–14, 1997.

[29] R. Draves, C. King, S. Venkatachary, and B. D. Zill. Constructing Optimal IP Routing Tables. In *Proc. IEEE INFOCOM*, 1999.

[30] W. Eatherton, G. Varghese, and Z. Dittia. Tree bitmap: hardware/software ip lookups with incremental updates. *SIGCOMM Comput. Commun. Rev.*, 34(2):97–122, Apr. 2004.

[31] D. Farinacci, V. Fuller, D. Meyer, and D. Lewis. Locator/ID Separation Protocol (LISP). Work in Progress, `http://tools.ietf.org/html/draft-farinacci-lisp-12`, Mar. 2009.

[32] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, FOCS '00, pages 390–, Washington, DC, USA, 2000. IEEE Computer Society.

[33] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 3(2):20, 2007.

[34] P. Francis, X. Xu, H. Ballani, D. Jen, R. Raszuk, and L. Zhang. FIB Suppression with Virtual Aggregation. Work in Progress, `http://tools.ietf.org/html/draft-francis-intra-va-01`, Oct. 2009.

[35] P. Francois, C. Filsfils, J. Evans, and O. Bonaventure. Achieving sub-second igp convergence in large ip networks. *SIGCOMM Comput. Commun. Rev.*, 35(3):35–44, July 2005.

[36] V. Fuller. Scaling Issues with Routing+Multihoming. `http://www.vaf.net/~vaf/apricot-plenary.pdf`, 1996.

[37] K. Gadkari, D. Massey, and C. Papadopoulos. Dynamics of prefix usage at an edge router. In *Proceedings of the 12th international conference on Passive and active measurement*, PAM'11, 2011.

[38] J. Hasan and T. N. Vijaykumar. Dynamic pipelining: making ip-lookup truly scalable. *SIGCOMM Comput. Commun. Rev.*, 35(4):205–216, Aug. 2005.

[39] W.-K. Hon, R. Shah, and J. S. Vitter. Compression, indexing, and retrieval for massive string data. In *Proceedings of the 21st annual conference on Combinatorial pattern matching*, CPM'10, pages 260–274, Berlin, Heidelberg, 2010. Springer-Verlag.

[40] I. Ioannidis and A. Grama. Level compressed dags for lookup tables. *Computer Networks*, 49(2):147–160, 2005.

[41] D. Jen, M. Meisel, D. Massey, L. Wang, B. Zhang, and L. Zhang. APT: A Practical Tunneling Architecture for Routing Scalability. Technical Report 080004, UCLA, 2008.

[42] D. Jen, M. Meisel, H. Yan, D. Massey, L. Wang, B. Zhang, and L. Zhang. Towards A Future Internet Architecture: Arguments for Separating Edges from Transit Core. In *ACM Workshop on Hot Topics in Networks*, 2008.

[43] E. M. Karpilovsky. Reducing Memory Requirements for Routing Protocols (thesis). Technical Report TR-861-09, Princeton University, 2009.

[44] J. Katajainen and E. Mäkinen. Tree compression and optimization with applications. *Int. J. Found. Comput. Sci.*, 1(4):425–448, 1990.

[45] V. Khare, D. Jen, X. Zhao, Y. Liu, D. Massey, L. Wang, B. Zhang, and L. Zhang. Evolution towards global routing scalability. *IEEE Journal on Selected Areas in Communications*, 28(8):1363–1375, Oct. 2010.

[46] C. Kim, M. Caesar, A. Gerber, and J. Rexford. Revisiting route caching: The world should be flat. In *Proceedings of the 10th International Conference on Passive and Active Network Measurement*, PAM '09, 2009.

[47] L. Li, D. Alderson, W. Willinger, and J. Doyle. A first-principles approach to understanding the Internet's router-level topology. In *Proc. of ACM SIGCOMM*, 2004.

[48] Q. Li, D. Wang, M. Xu, and J. Yang. On the scalability of router forwarding tables: Nexthop-Selectable FIB aggregation. In *Proc. IEEE INFOCOM*, 2011.

[49] T. Li. Preliminary Recommendation for a Routing Architecture. RFC 6115, 2011.

[50] H. Liu. Routing prefix caching in network processor design. In *in Tenth International Conference on Computer Communications and Networks*, 2001.

[51] Y. Liu, B. Zhang, and L. Wang. BGPMon-Next Generation BGP Monitor. `http://bgpmon.netsec.colostate.edu/f`.

[52] Y. Liu, B. Zhang, and L. Wang. Fast Incremental FIB Aggregation (FIFA). `http://www.cs.memphis.edu/~lanwang/fifa-tr.pdf`.

[53] Y. Liu, X. Zhao, K. Nam, L. Wang, and B. Zhang. Incremental forwarding table aggregation. In *Proc. IEEE Globecom*, 2010.

[54] V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Trans. Algorithms*, 4(3):32:1–32:38, July 2008.

[55] A. J. McAuley and P. Francis. Fast routing table lookup using cams. In *INFOCOM'93. Proceedings. Twelfth Annual Joint Conference of the IEEE Computer and Communications Societies. Networking: Foundation for the Future, IEEE*, pages 1382–1391. IEEE, 1993.

[56] 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):69–74, 2008.

[57] D. Meyer, L. Zhang, and K. Fall. Report from the IAB Workshop on Routing and Addressing. *RFC 4984*, 2007.

[58] D. R. Morrison. PATRICIA-practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, Oct. 1968.

[59] J. Moy. OSPF Version 2. RFC 2328, SRI Network Information Center, September 1998.

[60] G. Navarro and V. Mkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39:2007, 2006.

[61] S. Nilsson and G. Karlsson. Fast address lookup for internet routers. In *IEEE Broadband Communications*, pages 11–22, 1998.

[62] S. Nilsson and G. Karlsson. Ip-address lookup using lc-tries. *IEEE J.Sel. A. Commun.*, 17(6):1083–1092, Sept. 2006.

[63] R. Raman, V. Raman, and S. R. Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Algorithms*, 3(4), Nov. 2007.

[64] R. Raszuk, J. Heitz, A. Lo, L. Zhang, and X. Xu. Simple Virtual Aggregation (S-VA). Work in Progress, `http://tools.ietf.org/search/draft-ietf-grow-simple-va-10`.

[65] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol (BGP-4). *RFC 4271*, Jan. 2006.

[66] G. Rétvári, J. Tapolcai, A. Korösi, A. Majdán, and Z. Heszberger. Compressing ip forwarding tables: Towards entropy bounds and beyond. 2013.

[67] J. Rexford, J. Wang, Z. Xiao, and Y. Zhang. BGP routing stability of popular destinations. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, IMW '02, 2002.

[68] E. Silva de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Trans. Inf. Syst.*, 18(2):113–139, Apr. 2000.

[69] K. Sklower. A tree-based packet routing table for berkeley unix. In *USENIX Winter*, volume 1991, pages 93–99, 1991.

[70] H. Song, M. Kodialam, F. Hao, and T. Lakshman. Scalable ip lookups using shape graphs. In *Network Protocols, 2009. ICNP 2009. 17th IEEE International Conference on*, pages 73–82. IEEE, 2009.

[71] H. Song, J. Turner, and J. Lockwood. Shape shifting tries for faster ip route lookup. In *Proceedings of the 13TH IEEE International Conference on Network Protocols*, ICNP '05, pages 358–367, Washington, DC, USA, 2005. IEEE Computer Society.

[72] V. Srinivasan and G. Varghese. Faster ip lookups using controlled prefix expansion. In *ACM SIGMETRICS Performance Evaluation Review*, volume 26, pages 1–10. ACM, 1998.

[73] V. Srinivasan and G. Varghese. Fast address lookups using controlled prefix expansion. *ACM Transactions on Computer Systems (TOCS)*, 17(1):1–40, 1999.

[74] S. Stergiou and J. Jain. Optimizing routing tables on systems-on-chip with content-addressable memories. In *System-on-Chip, 2008. SOC 2008. International Symposium on*, pages 1–6. IEEE, 2008.

[75] A. Tariq, S. J. Tariq, and J. A. Uzmi. Taco: Semantic equivalence of ip prefix tables. International Conference on Computer Communications and Networks (ICCCN), 2011.

[76] G. Trotter. Terminology for Forwarding Information Base (FIB) based Router Performance. RFC 3222, 2001.

[77] J. Turner, G. Varghese, and M. Waldvogel. Scalable high speed ip routing lookups, Jan. 25 2000. US Patent 6,018,524.

[78] H. Tzeng. Longest prefix search using compressed trees. In *Proc. Globecom98*. Citeseer, 1998.

[79] Z. A. Uzmi, M. Nebel, A. Tariq, S. Jawad, R. Chen, A. Shaikh, J. Wang, and P. Francis. SMALTA: practical and near-optimal FIB aggregation. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, CoNEXT '11, 2011.

[80] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. *Scalable high speed IP routing lookups*, volume 27. ACM, 1997.

[81] W. Zhang, J. Bi, J. Wu, and B. Zhang. Catching popular prefixes at as border routers with a prediction based method. *Comput. Netw.*, 56(4):1486–1502, Mar. 2012.

[82] X. Zhao, Y. Liu, L. Wang, and B. Zhang. On the Aggregatability of Router Forwarding Tables. In *Proc. IEEE INFOCOM*, 2010.

[83] X. Zhao, D. J. Pacella, and J. Schiller. Routing scalability: an operator's view. *IEEE Journal on Selected Areas in Communications*, 28(8):1262–1270, Oct. 2010.

[84] N. Ziviani, E. S. de Moura, G. Navarro, and R. Baeza-Yates. Compression: A key for next-generation text retrieval systems. *IEEE Computer*, 33(11):37–44, 2000.