Computer Networks 83 (2015) 85–99

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Efficient FIB caching using minimal non-overlapping prefixes ☆

Yaoqing Liu [a,*], Vince Lehman [b], Lan Wang [b]

[a] Department of Computer Science, Clarkson University, Potsdam, NY 13699, USA
[b] Department of Computer Science, University of Memphis, Memphis, TN 38152, USA

ARTICLE INFO

Article history:
Received 15 September 2014
Received in revised form 13 January 2015
Accepted 3 March 2015
Available online 10 March 2015

Keywords:
FIB caching
OpenFlow
Routing
Scalability
Cache-hiding

ABSTRACT

The size of the global Routing Information Base (RIB) has been increasing at an alarming rate. As a direct effect, the size of the global Forwarding Information Base (FIB) has experienced rapid growth. This increase raises serious concerns for Internet Service Providers (ISPs) as the FIB memory in line cards is much more expensive than regular memory modules, so frequently increasing this memory capacity for all the routers is prohibitively costly to an ISP. Previous research on Internet traffic indicates that a very small number of popular prefixes receive most of the Internet's traffic, making "caching" a possible solution to reduce the FIB size. However, FIB caching may cause a *cache-hiding* problem where a packet's longest-prefix match in the cache differs from that in the full FIB, and thus the packet will be forwarded to the wrong next hop. Motivated by these observations, we propose an efficient FIB caching scheme that stores only non-overlapping FIB entries into the fast memory (i.e., a *FIB cache*), while storing the complete FIB in slow memory. Our caching scheme achieves a considerably higher hit ratio than previous approaches while preventing the *cache-hiding* problem. It can also handle cache misses, cache replacement, and routing updates efficiently. Moreover, we have implemented the proposed caching scheme using the OpenFlow platform, which allows a local or remote route controller to manage routes in the cache. We use real traffic of a regional ISP and a Tier1 ISP to carry out our experiments. Our simulation results show that with only 20 K prefixes in the cache (5.28% of the actual FIB size), the hit ratio of our scheme is higher than 99.95%. Our OpenFlow implementation achieves a hit ratio near 99.94%, which approaches the performance of the simulated results.

1. Introduction

In recent years, the routing table in default-free routers has been growing rapidly due to a variety of factors such as an increasing number of edge networks, increased use of multihoming, and finer-grained traffic engineering practices [2]. Both academic and industry communities [3–7] have been working on solutions to this problem. A direct consequence of the routing table (RIB) growth problem is the rapid growth of the forwarding table (FIB), which is calculated based on the RIB and looked up during packet forwarding. Although both trends are disturbing, Internet Service Providers (ISPs) are more concerned about the FIB size [8], because the FIB memory in line cards costs much more than the memory in route 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). To scale well with the increasing FIB size, a naive solution

☆ This work was partially supported by NSF Grant 0721645, USA. This paper is an extension of our previous publication "Efficient FIB caching using minimal non-overlapping prefixes" [1] that appeared in SIGCOMM Computer Communication Review.

* Corresponding author.
E-mail addresses: liu@clarkson.edu (Y. Liu), vslehman@memphis.edu (V. Lehman), lanwang@memphis.edu (L. Wang).

http://dx.doi.org/10.1016/j.comnet.2015.03.003
1389-1286/

is to add more memory to the routers. However, it is challenging to meet the continuously changing memory requirements and, as FIB memory is expensive, upgrading the memory of all the routers of an AS is not compelling for ISPs. Moreover, as the size of FIB memory increases, the line card may become more power-hungry and the lookup time may also increase for non-TCAM-based FIB implementations [2].

In the long run, modifying the current routing architecture and protocols seems to be the best solution to these problems [4]. However, such proposals may take a long time to deploy due to the high costs associated with them. Meanwhile, ISPs cannot afford to upgrade all their routers frequently. 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 [9].

One approach to reducing the impact of large FIBs is to use high-speed memory as a *cache* to store the most popular routes [10–13] 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 [11,13–15] have shown that a small number of popular prefixes contribute to most of the observed traffic. The data traces from a regional ISP and a tier-1 ISP used in our evaluation also support 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 match rather than exact match. If not well designed, 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, and thus the packet will be forwarded to the wrong next hop (Section 2.1). 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 develop algorithms to systematically handle cache initialization, cache misses, cache replacements, and routing updates. Following this design, we implemented our FIB caching scheme using the OpenFlow platform to control the FIB cache. When a cache miss occurs, our OpenFlow controller generates the matching non-overlapping leaf prefix and installs it in the FIB cache in an OpenFlow switch. Lacking efficient real-time access to the least recently used or least frequently used flow entry in the FIB cache, our controller infers the importance of a prefix based on how frequently a cache miss happens on the prefix and adjusts the cache timeout on the prefix dynamically to reduce cache misses.

Our simulation results show that, for a routing table of 378 K prefixes, our scheme achieves an average hit ratio higher than 99.95% using a cache size of 20 K prefixes (5.28% of the full FIB size), and our scheme outperforms alternative proposals in terms of hit ratio and number of FIB updates. Our OpenFlow implementation achieves a hit ratio near 99.94%, which approaches the performance of the simulated results. In addition, we fill the *initial* empty cache with the *shortest* non-overlapping prefixes generated from the full FIB, which significantly increases the hit ratio for the initial traffic. Our simulation results show that the initialized cache has a hit ratio of 85% for the first 100 packets, compared to 65% for an uninitialized cache (i.e., an empty cache at the beginning).

The remainder of the paper is structured as follows: Section 2 gives an overview of our scheme. Section 3 presents our FIB caching algorithm in detail. Section 4 describes our OpenFlow design and implementation. Section 5 evaluates our scheme and compares it with other approaches. Section 6 discusses related work and Section 7 concludes the paper.

2. Design overview

Fig. 1 illustrates a router architecture with the proposed FIB caching scheme. The control plane contains the RIB while the Slow FIB and FIB cache reside in the data plane. The Slow FIB memory contains a copy of the full

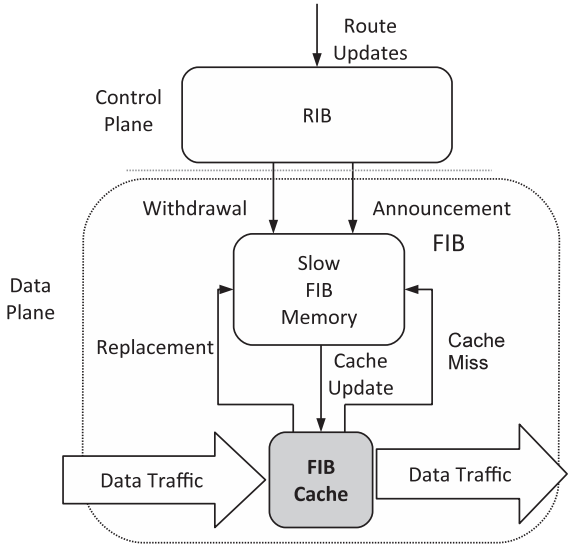


Fig. 1. Design architecture for FIB caching.

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: Route Announcement, Route Withdrawal, cache miss and cache replacement. The Route Announcement and Route Withdrawal events correspond to changes in the RIB, which need to be propagated to the FIB. A Cache Miss event occurs when an incoming packet does not have a matching prefix in the FIB cache, and a cache replacement event occurs when the FIB 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 FIB Cache, we explain the cache-hiding problem and outline our solution for handling it in the following section.

2.1. 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 1, 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 1(b)). Now, suppose another data packet destined to 10010001 arrives. Then, 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 the 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 1001/4 "hides" the more specific prefix 100100/6 in the full FIB.

Table 1
FIB entries and cache entries.

Label	Prefix	Next hop
(a) FIB entries		
A	10/2	4
B	1001/4	2
C	100100/6	1
(b) Cache entries		
B	1001/4	2

Note: The cache is initially empty and receives one entry upon the first cache miss.

2.2. Our solution to cache-hiding

To illustrate our solution, we use Patricia Tries (i.e., Radix Tree) [16] 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.

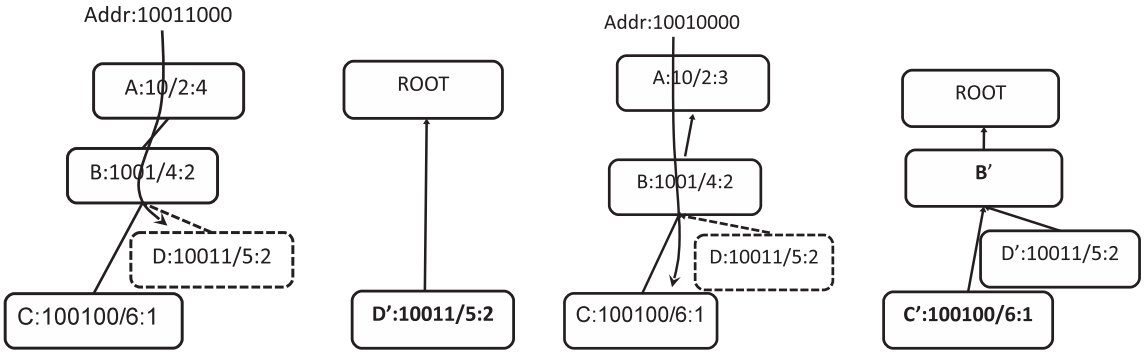
We cache the most specific *non-overlapping* prefixes that do not hide any longer prefixes in the full FIB to avoid the cache hiding problem. In Table 1, *C*'s address space is covered by *B*, so they are *not* non-overlapping prefixes (see Fig. 2(a)). As such, we cannot simply load the prefix *B* (1001/4) 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* (Fig. 2(a)) and put it into the cache (Fig. 2(b)). *D* (10011/5) has the next hop 2, which is the 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 (Fig. 2(c)). The matching prefix is then loaded into the cache (Fig. 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 number of cached prefixes.

In addition to solving the cache hiding problem, our proposal may also simplify hardware design for the cache and make the cache more efficient. First, since the interdependency between prefixes is eliminated, there should be only one matching prefix, if any, for a data packet. A TCAM-based (Ternary Content-Addressable Memory) cache then does not need to sort the cached prefixes in decreasing order of prefix lengths, thus making any cache update operation faster. Second, the priority encoder logic used to select the longest matching prefix can also be removed [17].

2.3. Workflow for handling data traffic

Fig. 3 shows how our design handles an incoming packet. In the 'Init' (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 (Section 3.6).

For any incoming packet, a prefix match is performed on the Patricia Trie of the cache (note that there is no need for longest prefix match since the prefixes in the cache do not overlap). In the case of a match, the packet is forwarded accordingly. In the case of a *cache miss*, a longest prefix match 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 in the Slow FIB 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). Packets that experience a cache miss can be stored in a separate queue and forwarded once the prefixes from Slow FIB memory are installed into the cache.



(a) Generate prefix D in Full FIB

(b) Insert D' in FIB Cache

(c) Select prefix C in Full FIB

(d) Insert C' in FIB Cache

Fig. 2. Selection or generation of a leaf prefix.

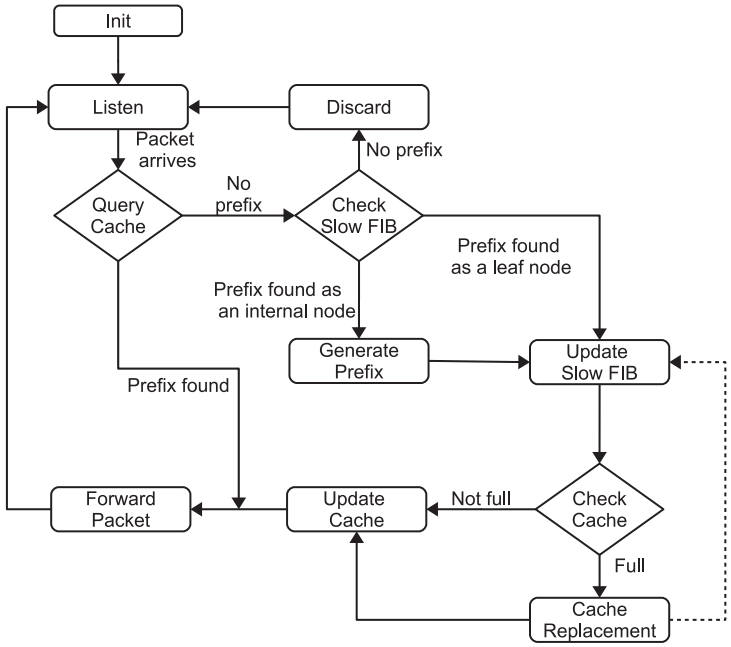


Fig. 3. Workflow for Handling Incoming Data Traffic (the dotted line means that during cache replacement, the Slow FIB needs to be updated but the flow of operation does not continue beyond that point).

When a new prefix needs to be installed into a *full* cache, one of the existing prefixes in the cache needs to be removed (Section 3.3) using a suitable cache replacement mechanism. Subsequently, the corresponding entry in the full FIB may be deleted or updated according to different scenarios that will be discussed in Section 3.3.

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 Sections 3.4 and 3.5.

3. Design description

In Section 2, we highlighted the operations taking place at each component of the caching model. This section provides the details of each operation and also discusses more complex scenarios.

3.1. Data structure

Each node in the Patricia Trie of the Slow FIB is one of four types that may change upon an update. These types help us to keep the FIB cache, Slow FIB, and 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.

3.2. Handling cache misses

In the case of a cache miss, we perform a longest prefix match 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*, 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, then 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. A new glue node with prefix Y will be inserted below node P with the same next hop as P. The new leaf node to be installed into the cache will be a child of the new glue node, with a prefix equal to the first $l+1$ bits of X, a next hop inherited from the parent, and a type of *CACHE_ONLY* as it is generated on demand.

Now we provide a detailed example of case *c* mentioned above. In Fig. 4(a), the matching node B with prefix 1001/4 has both a left child (C) and a right child (D). According to the rules above, Y_L is the common portion of X (10010100) and C (100100/6), with a value 10010/5 and Y_R is the common portion of X and D (10011/5) with a value 1001/4. Therefore, we pick the longer one 10010/5 for Y, and l is the prefix length of Y, i.e., 5. The new leaf node (F) has a prefix of $X/(l+1)$, i.e., 100101/6, a next hop of 2, and a node type of *CACHE_ONLY* as shown in Fig. 4(c). Fig. 4(g) and (h) show the cache entries before and after the update.

3.3. Handling cache replacement

When the cache becomes full and a new prefix needs to be inserted, an existing prefix in the FIB cache needs to be evicted according to the replacement strategy (e.g. LRU). We first remove the existing prefix from the cache, insert the new prefix into the cache, and then update the Slow FIB. There are two cases that may occur in the event of prefix removal:

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 from the cache;
2. if the corresponding node type is *FIB_CACHE*, it means that this node has a corresponding prefix in the RIB so we cannot remove it from the FIB. Therefore, we change the type to *FIB_ONLY*.

Fig. 5 shows the removal of prefix 100100/6 from the cache (cache replacement event). Fig. 5(a) and (b) show the cache operations. Fig. 5(c) and (d) show the Slow FIB operations.

3.4. Handling Route Announcements

A Route Announcement may add a new prefix or update an existing entry in the Slow FIB (see the complete workflow in Fig. 6). Below we describe each scenario briefly.

When adding a new node to the FIB trie, we need to handle the following two cases (Fig. 6):

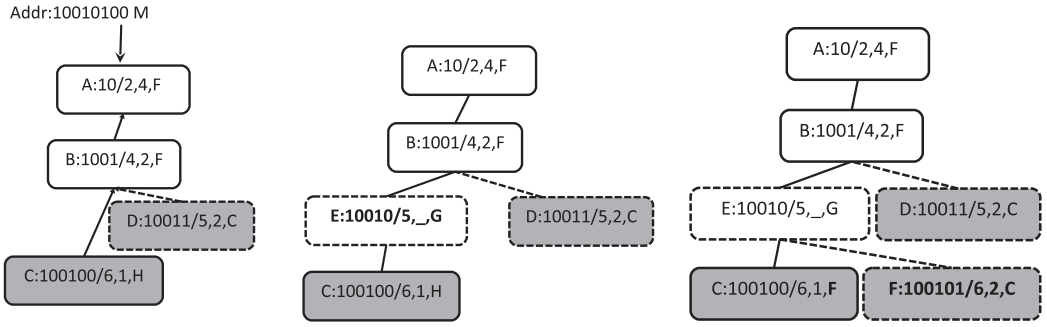
1. The new node is added as a leaf node: if its direct parent node type is *CACHE_ONLY* (i.e., the prefix of this node was generated on demand and is in the cache), then we remove the parent node from both the FIB and the cache in order to avoid the cache-hiding problem. If the direct parent of the new 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, as illustrated in Fig. 6.

3.5. Handling route withdrawals

For a Route Withdrawal, the matching node can be either a leaf node or an internal node, and we process it as follows (Fig. 6):

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



(a) FIB trie before update (b) FIB trie during update (c) FIB trie after update

Node : Prefix	Next Hop	Type
A : 10/2	4	F
B : 1001/4	2	F
C : 100100/6	1	H
D : 10011/5	2	C

(d) FIB entries before update

Node : Prefix	Next Hop	Type
A : 10/2	4	F
B : 1001/4	2	F
C : 100100/6	1	H
D : 10011/5	2	C
E : 10010/5	_	G

(e) FIB entries during update

Node : Prefix	Next Hop	Type
A : 10/2	4	F
B : 1001/4	2	F
C : 100100/6	1	F
D : 10011/5	2	C
E : 10010/5	_	G
F : 100101/6	2	C

(f) FIB entries after update

Node : Prefix	Next Hop
C' : 100100/6	1
D' : 10011/5	2

(g) Cache entries before update

Node : Prefix	Next Hop
C' : 100100/6	1
D' : 10011/5	2
F' : 100101/6	2

(h) Cache entries after update

Fig. 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 RIB. A dashed rectangle denotes a generated node due to a cache miss update. A gray node denotes a node in the cache.

only leaf nodes into 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 hop was derived from this node. Finally, we update the cache accordingly.

3.6. Cache initialization

Handling initial data traffic is a major concern for deploying caching mechanisms [9]. To address this issue, we fill up the initial empty cache with a set of leaf prefixes from the FIB that cover the most IP addresses. More specifically, a breadth-first search is employed to find the shortest leaf prefixes from the full FIB Trie (up to the cache size). This way we achieve a high hit ratio while avoiding the cache-hiding problem. In addition, if a router is able to store the cache content in non-volatile memory before restarting, it can use this stored cache to fill up the initial cache.

4. OpenFlow implementation

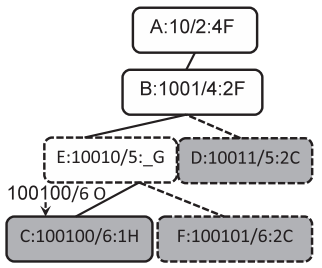
To evaluate the efficiency and effectiveness of our FIB caching algorithms in a realistic setting, we need a router that supports a FIB cache and a mechanism to control the forwarding entries in the cache. We chose the OpenFlow platform [18] as it allows us to experiment with programmable switches without having to modify existing hardware or software routers extensively. Note, however, that this is not the only option for networks to deploy FIB

Node : Prefix	Next Hop
C' : 100100/6	1
D' : 10011/5	2
F' : 100101/6	2

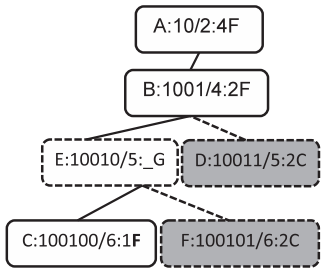
(a) Cache entries before update

Node : Prefix	Next Hop
D' : 10011/5	2
F' : 100101/6	2

(b) Cache entries after update



(c) FIB before update



(d) FIB after update

Fig. 5. Example of prefix removal during cache replacement update.

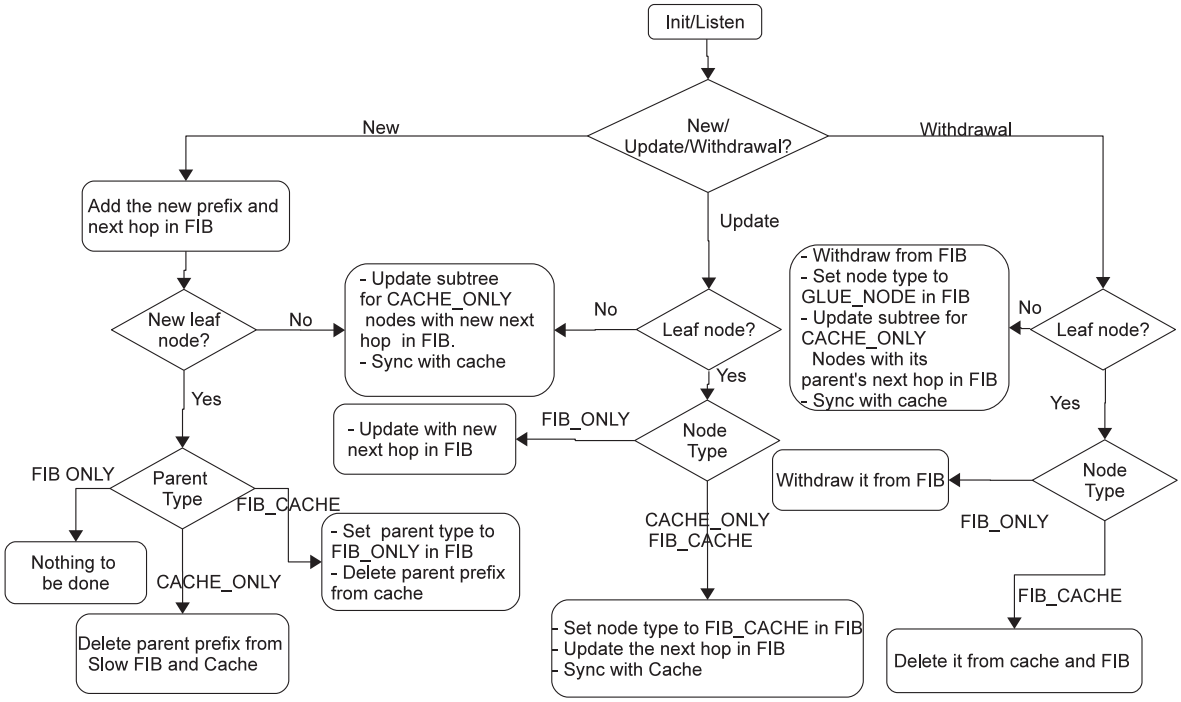


Fig. 6. Workflow for handling announcements and withdrawals (loopbacks to the 'Listen' state are not shown).

caching. In practice, ISPs could choose to deploy either (a) routers that have a full FIB in slow memory and a FIB cache in fast memory if such routers exist; or (b) separate controllers and switches as in OpenFlow. In the first case, the delay incurred by cache misses is minimal as the full FIB is located in the same router except in slower memory. In the second case, the controllers do not have to be physically distant from the switch. In regional ISPs, there can be a few controllers in the ISP each serving routers in nearby areas. In large ISPs, one or two controllers can be placed in each Point of Presence to minimize the delay between the controller and the routers (they may very likely be located in the same room) (see Fig. 7).

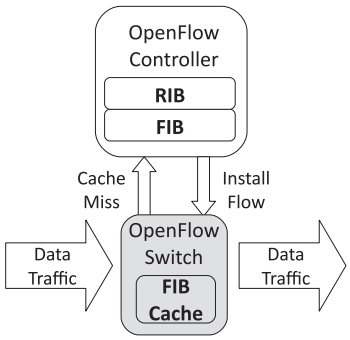


Fig. 7. Design architecture for OpenFlow implementation.

4.1. OpenFlow switch

An OpenFlow switch maintains a flow table with corresponding actions for each flow entry and processes any incoming, matching packets based on those actions. The switch communicates with an external controller that can install flow entries in the switch's flow table, which allows researchers to define desired behavior for the switch without explicitly accessing or programming the switch [18]. Many hardware switches exist already [19] that support various versions of the OpenFlow specification, a generic standard that defines certain required features while detailing other optional features. As of OpenFlow Switch Specification 1.4 [20], Layer 3 operations such as the ability to perform TTL decrementing and MAC address updates are defined as optional, thus it is up to the hardware implementation to provide support for such actions. For example, the HP 3800 Switch Series provides OpenFlow 1.3 support with the previously mentioned layer 3 actions and the ability to hold 64 K flow entries [21].

4.2. OpenFlow controller

We use the OpenFlow controller to maintain the RIB and full FIB, and the OpenFlow switch as a FIB cache.[1] The controller performs the algorithms on the full FIB mentioned above in Section 3. The controller and switch may communicate over a secure connection if the hardware implementation supports this feature (see Section 6.3.3 of the OpenFlow Switch Specification 1.4 [20]). The overhead of a secure connection depends on the encryption algorithm, hardware speed and other implementation specific parameters; the reader is referred to a recent study on SSL processing overhead by Zhao et al. [22]. We note that the major overhead of SSL has been evaluated to be during session establishment [22]. Since the OpenFlow components remain connected during operation, the session establishment overhead is typically incurred only once at the beginning. In case the controller and switch are disconnected, SSL session re-negotiation mechanism can greatly reduce the overhead of re-establishing the secure connection [22]. Furthermore, our evaluation results (Section 5) show that the cache miss ratio of our algorithm is extremely low, e.g., on the order of 10's of cache misses per second per switch, so the communication between the controller and switch is infrequent.

When the controller first connects with the switch, the FIB and RIB patricia tries are built and an initialized cache is generated. After the initial cache entries are calculated, the controller installs a flow entry for each prefix; the flow entry defines its match rule so that any incoming packets that hit the initialized prefix are forwarded according to that flow entry's actions. The controller also maintains an update file that contains the announcement and withdrawal of routes; the controller processes these updates and applies them to the RIB. On the event that a packet misses the switch's flow table, the switch notifies the controller by sending information about the packet that caused the miss. The controller uses the packet's destination address to generate a flow entry and installs the flow entry in the switch. The switch then uses the flow entry to forward the packet.

4.3. Cache management

In OpenFlow, cache replacement is currently achieved by assigning idle timeout values to flow entries. The controller installs flow entries in the switch using the algorithm in Section 4.3.1 to assign an idle timeout value, and the switch removes any flow entries that exceed their idle time.

We have found that it is infeasible to implement better cache replacement algorithms such as LRU or LFU based on the current OpenFlow specification. An LRU algorithm requires the controller to know which flow entry has been idle for the longest time. While a switch does maintain the time a flow entry has been idle, the controller cannot request the idle time information. Moreover, even if the controller can request every flow entry's idle time, it will have to get all the flow entries' idle time to find the one with the *longest* idle time, which is very inefficient. Note that the controller can receive notifications containing a flow entry's idle time when a flow entry is removed, but knowing a removed flow entry's idle time is not useful for picking another flow entry for removal when the controller has a new flow entry to install. An LFU algorithm requires the controller to know which flow entry has the *lowest* packet count, but the controller is unable to request this information directly. It must request the entire flow table and choose a single flow entry with the lowest packet count. In our experiments, it took around five seconds for the controller to receive the complete stat reply (containing statistics for the entire flow table) from a switch with 20 K flow entries installed in the flow table, an impractical amount of time to wait due to the fact that miss events are continuously sent to the controller for processing. Other research also shows long delays when collecting flow table statistics from an OpenFlow switch [23]. OpenFlow's usefulness for research and experimentation would be greatly increased with the inclusion of switch defined cache replacement methods and flow entry requests based on longest idle time or lowest packet count.

As OpenFlow does not provide a means besides flow entry timeout for cache replacement, we decided to design

[1] Our code is available for research and other non-commercial purposes. Please contact us to obtain a copy of the source code.

an algorithm that uses a variable timeout to maintain a reasonable cache size.

4.3.1. Timeout algorithm

Maintaining a modest cache size is critical to the success of an OpenFlow implementation; a large cache size contradicts the purpose of FIB caching. Flow entries should exist in the cache for as long as they are needed and then remove themselves to make room for more useful flow entries. Thus, it was necessary to determine suitable timeout values to assign to flow entries so as to minimize the cache size while maximizing the hit-ratio. To produce potential timeout values, we ran a simulation where the switch's cache size was unlimited and the use of each prefix was recorded. Using these recorded prefix access-times, we can determine the times when a prefix was unused (idle). We first calculated the quartiles for each individual prefix and then calculated the quartiles of the combined previously calculated quartiles. The results from our 24-hour traffic trace showed that the median of the prefix timeout medians was 43 seconds. We used this value as our initial timeout value for newly installed prefixes. But, a single timeout value is not enough as many useful prefixes could be idle for this amount of time, removed from the cache, and then accessed again in a short period of time. This premature removal would cause a cache miss and negatively impact the cache's hit-ratio. Therefore, prefixes that are frequently removed and installed in a short period of time should be installed with a greater timeout value that increases multiplicatively with each subsequent removal and quick re-installation.

Based on previously collected data and through experimentation, we designed an algorithm that attempts to minimize the cache size while maximizing the hit-ratio using idle timeouts to remove flow entries from the cache. The algorithm maintains each prefix that has been installed as well as whether the prefix is currently in the cache, a recent removal timer that starts on the prefix's last removal from the cache, and the number of times the prefix has been removed from the cache.

The recent removal timer associated with each prefix begins decrementing from 60 s when the prefix is removed from the cache. If the recent removal timer reaches 0 s and the prefix has not been installed again, the prefix's removal count is reset to zero. This clears the history for the prefix and effectively marks the prefix as unpopular. Otherwise, if the prefix is reinstalled in the cache before the recent removal timer reaches 0 s, the timer is stopped because the prefix is considered popular again.

On the installation of a prefix, the algorithm checks whether the prefix has been installed previously; if not, the prefix is installed with a timeout of 43 s. If the prefix has been installed previously, the controller checks the prefix's removal count. A prefix with a removal count of one means the prefix is being reinstalled during the 60-second period and so is installed with a longer timeout value. If the prefix has a removal count greater than one, the prefix is installed with its previous timeout multiplied by two. Otherwise, the prefix is installed with a 43 s timeout. Our longer timeout value, 1100 s, was chosen from experimentation that kept the cache size under 20,000 flow entries while maintaining a steady flow entry count. This process is illustrated in Algorithm 1 in Appendix A.

When a prefix in the cache remains unused for its idle timeout period, it is removed from the cache and the controller is notified. The controller retrieves the entry for the corresponding prefix, sets the prefix entry as not in the cache, increments the removal counter for that prefix, and starts the recent removal timer for that prefix. As time advances, the timers for recently removed prefixes advance until either the timer reaches 0 s or the prefix is reinstalled into the cache. When a prefix's recent removal timer reaches 0 s, the removal counter for that prefix is set to 0, which marks the prefix to be installed with the initial timeout value on its next installation. Algorithm 2 and 3 in Appendix A detail these procedures.

4.3.2. Comparison with a similar algorithm

After we independently designed the above variable timeout algorithm, we discovered another work [24] that pursued a similar process for managing flow entries in an OpenFlow switch. Vishnoi et al. proposed an algorithm that installs flow entries with an initial minimum idle timeout and increases subsequent idle timeouts for flow entries that are frequently evicted from the switch [24]. There are, however, a number of differences between their algorithm and ours. First, we use different metrics to set the initial timeout: their algorithm uses the 80th percentile of interarrival times from sample traffic traces to set an initial idle timeout value while our algorithm uses the median of the medians from the interarrival times of each prefix. Second, the idle timeout value is adjusted in different ways. Our algorithm sets a flow entry's idle timeout to an experimentally determined large value on its second installation and then multiplicatively increasing the timeout value on subsequent installations and our algorithm does not enforce a maximum idle timeout value. Moreover, their algorithm compares a flow entry's active time and idle time to reduce the idle timeout value assigned to short lived flow entries that are repeatedly removed whereas our algorithm only considers the frequency of flow entry removals when assigning an idle timeout value. Fine-tuning the timeout algorithm is one area of our future work. Nevertheless, our evaluation results in Section 5.7 show that our algorithm closely approximates the performance of the LRU algorithm, even though it is simpler than the algorithm proposed by Vishnoi et al.

5. Evaluation

In this section, we first introduce our methodology, then show the distribution of traffic to different destinations in our traffic traces, and finally present our simulation and implementation results.

5.1. Methodology

We evaluate our algorithm using a simulated FIB caching router with an ideal cache replacement algorithm and

an LRU replacement algorithm, as well as our OpenFlow implementation.

The ideal caching simulation gives us the optimal cache hit ratio, which cannot be achieved by any real-time cache replacement algorithm, by preprocessing the traffic trace to get the best prefix to replace at any given time. More specifically, we first simulate a FIB caching router with an unbounded cache size and record the access-times of each prefix. Then we use the prefix access-times to remove the flow entry that is the least useful at the current time (the flow entry that will be used furthest in the future). The simulation uses as input prefix access-times, a traffic trace, a routing table and routing updates. The results are reported in Section 5.7 to gauge the performance of our OpenFlow implementation.

A previous study conducted by Kim et al. [11] shows that the LRU algorithm performs almost as well as the ideal cache replacement algorithm. Since the latter is infeasible in practice, we simulated our cache with the LRU algorithm to obtain more realistic results. Our program takes a traffic trace, a routing table and routing updates as input to simulate packet lookup and forwarding process. The LRU algorithm is used in all the simulations in this section except those in Section 5.7.

Our OpenFlow implementation uses the design described in Section 4. We use a network emulator called Mininet [42] to create a virtual network with two hosts and an OpenFlow switch running Open vSwitch software [26]. One of the hosts sends packets from a traffic trace to the switch. For simplicity, the next hop of all the flow entries is set to the other host. This simplification does not affect our evaluation results since the specific next hop value does not have an impact on the hit ratio or overhead of FIB caching. The virtual network runs in a virtual machine with 2 GB RAM and a 100% CPU execution cap on a host machine with a 2.7 GHz Intel Xeon E5-2680 CPU. We use another machine with a 3.40 GHz Intel Pentium 4 CPU and 4 GB RAM to run an OpenFlow controller programmed using POX [27], an OpenFlow development platform that supports OpenFlow version 1.0 [28]. The controller and switch communicate over a TCP session established by POX. Since they are located on the same subnet, the one-way delay between them is very small (less than 1 ms). In order to keep the cache size below 20 K, the switch is started with an initial cache of 10 K prefixes; otherwise, the misses against the initial cache would cause the cache to grow larger than 20 K. The results from our OpenFlow implementation are described in Section 5.7.

We utilized two traffic traces to evaluate our scheme. The first one is a 24 h traffic trace of more than 4.1 billion packets from a regional ISP collected from 12/16/2011 to 12/17/2011. To validate the results from the first trace, we used a second trace from a very different source – a backbone router with a much higher forwarding speed in a Tier1 ISP. The data trace contains about 2.5 billion packets collected by CAIDA [29] from 13:00 to 14:00 on 06/19/2014. CAIDA collects only a one-hour trace each month from each monitor due to storage limitations. This trace is the most recent and best public traffic trace we can find. For the regional ISP data trace, we obtained routing tables and updates of 30 different routers from the route-views2 data archive [30] 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 one of them in most cases. Similar operations were performed over the second traffic trace using a global routing table on 06/19/2014 from RouteViews [30]. Note that all the following results were obtained using the first traffic trace, unless otherwise noted.

We also compared our scheme with different cache-hiding approaches, the most straightforward being the *Atomic Block* approach. This scheme loads not only a matching prefix into the cache but also finds all the sub-prefixes of the matching prefix in the FIB. It then loads them into the cache so that subsequent packets will not encounter the cache-hiding problem. *Uni-class*, another method, divides up a matching prefix into multiple fixed-length (24 bits) sub-prefixes on the fly and installs the one matching the destination address into the cache [11]. This approach assumes that 24 is the longest prefix length in the FIB so the cached and/24 are separated prefixes will not hide more specific prefixes in the FIB. This assumption is usually true as operators filter out prefixes longer than and /24 are separate to prevent route leaks. Moreover, we compared our approach with three techniques proposed by Liu [12], Complete Prefix Tree Expansion (CPTE), No Prefix Expansion (NPE) and Partial Prefix Tree Expansion (PPTE), using a static routing table. Finally, we compared the differences between our schemes and the RRC-ME (Reverse Routing Cache using Minimal Expansion Prefix) algorithm proposed by Akhbarizadeh and Nourani [10], which uses a binary tree (with no expansion) and only installs or generates a disjoint prefix into the cache on the fly.

5.2. Traffic distribution

Fig. 8 shows the traffic distributions over prefixes of two global routing tables corresponding to the two data traces. The *x*-axis represents the popular prefix rank, and

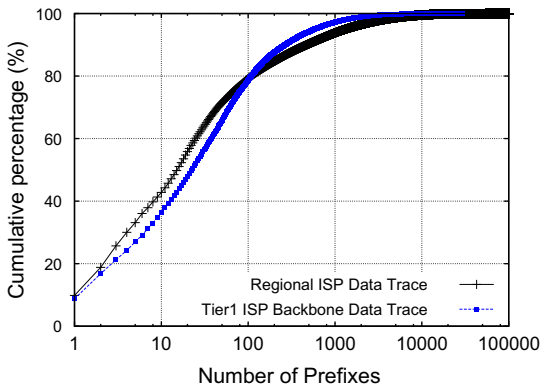


Fig. 8. Traffic distribution on non-overlapping prefixes.

the *y*-axis represents the cumulative percentage of the IP packets covered by the popular prefixes. For the regional ISP traffic distribution (black points line), we make two main observations: (a) the top 10, 100, 1 K, 10 K, 20 K 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 [14,11,15,13], i.e., a very small number of popular prefixes contributes 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 feasiblity of introducing an efficient caching mechanism for the routers. For the Tier1 ISP traffic distribution (blue dots line), the top 1 K, 10 K, 20 K popular prefixes cover about 97.25%, 99.96%, and 99.99%, respectively, of the traffic and thus yields much higher skewness than the regional one, which further justifies our observations.

5.3. 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 to evaluate a caching scheme. 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 1 K to 20 K prefixes. Fig. 9(a) shows different hit ratios for one router with five different cache sizes over the 24 h period of the regional ISP traffic trace. We observe that on average the hit ratio is 96.83%, 98.52%, 99.25%, 99.84%, 99.95% for the cache size of 1 K, 2 K, 3.5 K, 10 K and 20 K, respectively. The dips around 870 million data packets are due to the traffic pattern around 7:30 a.m. 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 as the cache size increases. Other routers have very similar results to this one.

Fig. 9(b) shows different hit ratios for one router with five different cache sizes over 1 h period of the Tier1 ISP traffic trace. Surprisingly, the Tier1 ISP traffic from a backbone router has higher hit ratios than the counterpart of the regional ISP traffic, which validates the feasibility and effectiveness of our approach. Overall, both traffic traces from different times and locations demonstrate similar properties in terms of high hit ratios and skewness.

Fig. 9(c) compares the hit ratios for the different caching approaches, such as Atomic Block approach and Uni-class approach, with a fixed cache size of 20 K. Our approach has a 99.95% hit ratio on average. The Atomic Block approach has a 99.62% hit ratio and the 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 5.6. 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 prefix; these 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.

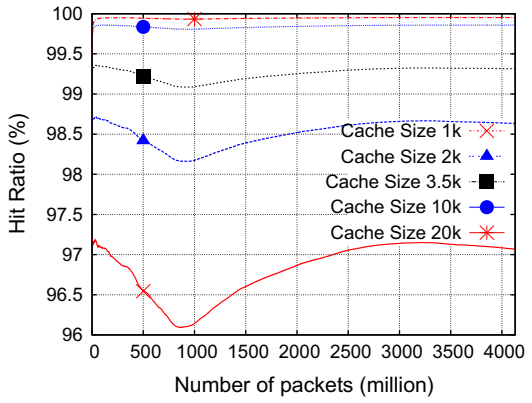
Moreover, we compared our approach with the three techniques proposed by Liu [12], CPTE, NPE and PPTE, using a static routing table (the author did not specify update handling algorithms). NPE does not increase the FIB size and has a 99.16% hit ratio on average. PPTE increases the FIB size by 13,384 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 6288 and reach a hit ratio of 99.95% on average. Finally, the RRC-ME algorithm proposed by Akhbarizadeh and Nourani [10] has the same hit ratio as our scheme (not shown in the figure), but our update handling algorithm is much more efficient (Section 5.5).

5.4. Initial traffic handling

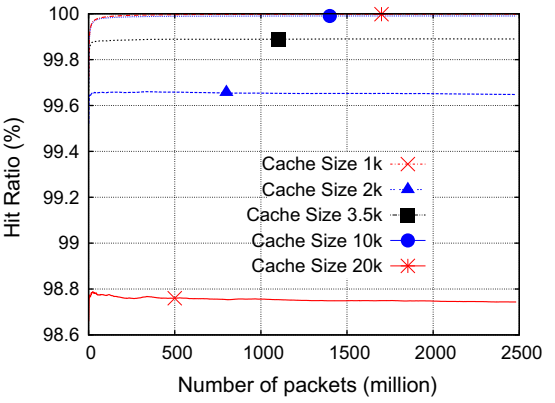
One of the biggest concerns for ISPs is how to handle the initial traffic when the cache starts with an empty set [9]. Instead of a cold start, we fill the initial empty cache completely with the shortest non-overlapping prefixes if there is no history of popular prefixes available. Fig. 9(d) shows the 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.

5.5. Routing update handling performance

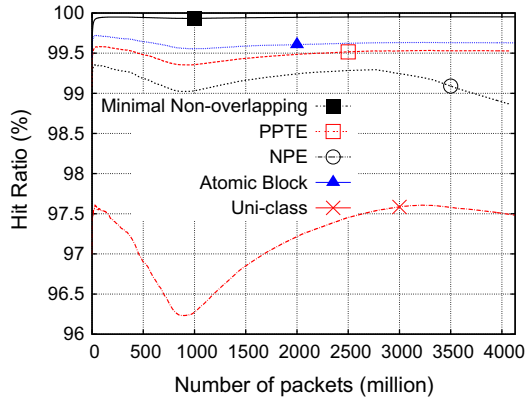
Fig. 10 shows the routing update handling performance. The top curve represents the total number of RIB updates. The middle curve represents the total number of updates (8348) pushed to the cache including next hop changes (8251) and prefix deletions. The bottom curve shows the number of prefix deletions (97), which is only 3.18% of the total number of RIB updates. Since we store a few flags in the routing table (control plane), such as *FIB_ONLY*, to infer whether certain prefixes need to be updated in the cache, actually very few updates are pushed to the cache, the updates have almost no influence on the cache hit ratio. On the other hand, the authors in the RRC-ME approach [10] did not utilize any control plane information, therefore, both zero padded and one padded IP numbers of the lookup prefix need to be used to guarantee its parent being present in RRC (Reverse Routing Cache). In



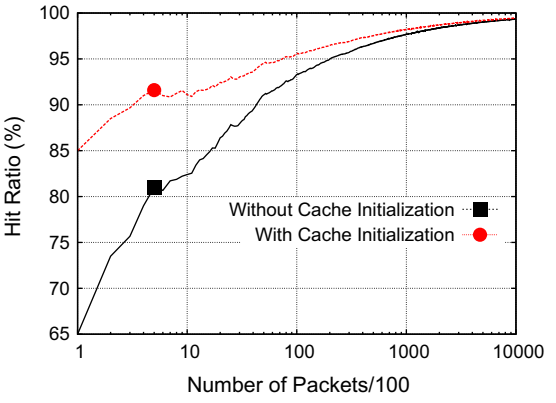
(a) Hit Ratios for One Regional ISP Router



(b) Hit Ratios for One Tier1 ISP Backbone Router



(c) Hit Ratios for Different Approaches



(d) Hit Ratios With and Without Cache Initialization

Fig. 9. Different hit ratios.

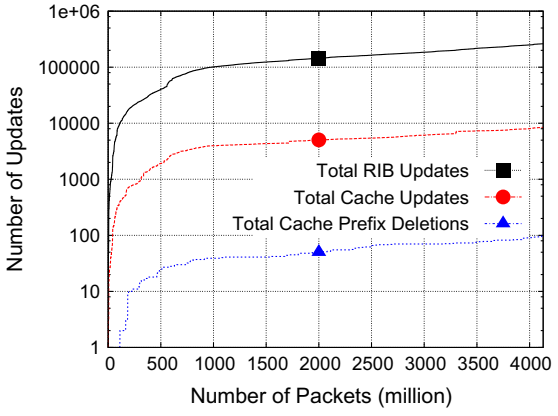


Fig. 10. Update handling impact.

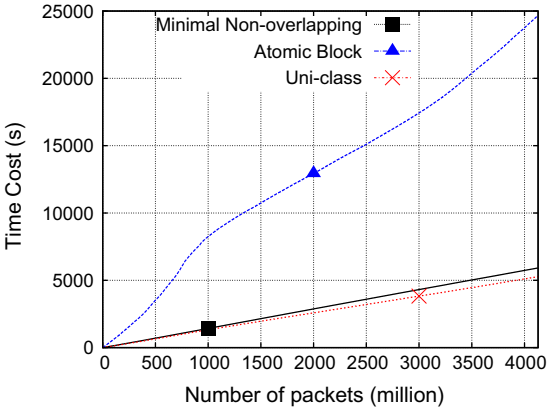


Fig. 11. Total time cost.

other words, each prefix that need to be updated must be converted into two IP addresses first and then looked up in the cache to find the matching parent prefix. In the process, the cache will be interrupted twice if there is no matching prefix; otherwise, the cache will receive three memory access requests. Specifically, in the period of 24 h, the previous work needs at least 523,754 (all together 261,877 updates) cache lookups as compared to our scheme which needs only 8251 lookups.

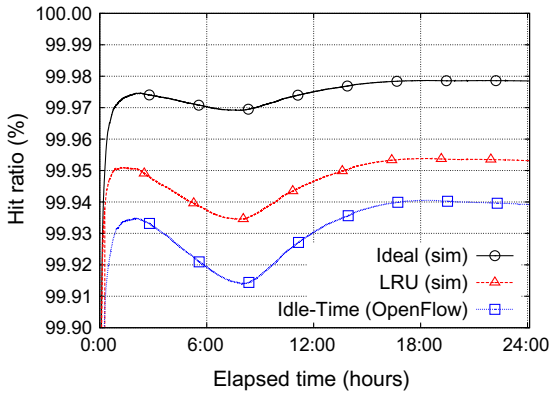


Fig. 12. OpenFlow evaluation.

5.6. Time cost

Fig. 11 compares the time cost to process all the routing updates and data of 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 has a much higher hit ratio as shown previously.

5.7. OpenFlow

To evaluate how well our implementation performs, we compare the hit-ratio between the OpenFlow implementation, an Least Recently Used (LRU) simulation and the ideal caching algorithm; Fig. 12 shows the results. The OpenFlow implementation closely resembles the hit-ratio of the LRU simulation which suggests the feasibility of a real world FIB caching implementation. The traffic trace used in our experiments contained around 4 billion packets collected over a 24 h period. This means, on average, there were around 50 K packets arriving at the switch per second. The OpenFlow results show an average hit-rate of 99.94%, which suggests approximately 31 packets will miss the switch's flow table per second. Thus, the OpenFlow controller must be able to handle an average of 31 packets per second per additional dependent switch added, which is not a high traffic load.

We also measured the additional delay when a cache miss occurred in our implementation by capturing packet miss messages sent from the switch to the controller and flow modification messages sent from the controller to the switch for installing new prefixes. We measured an average delay of under 20 ms per cache miss, which includes the delay in the POX platform that implements generic controller functionality and the delay in our own code for generating the prefix to install in the cache. Since our code showed an average processing time of less than 1 ms per cache miss, we suspect that the majority of the additional processing time may be incurred by the POX code in the controller. As POX is experimental software mostly used for research ("a platform for the rapid development and prototyping of network control software using Python" according to http://www.noxrepo.org/pox/about-pox/), it is not expected to achieve the performance of production systems in market today. We also expect that the performance of OpenFlow controllers will continue to improve, but this is out of scope for this work. Our future work will be to investigate this processing delay and if necessary, migrate our implementation to a more optimized controller platform.

6. Related work

Two main solutions can lead to FIB size reduction without changes to the routing architecture, FIB aggregation and FIB caching. FIB aggregation is to aggregate a large FIB table into a smaller one with the same forwarding results. There have been a number of FIB aggregation algorithms proposed in last few years [31–36]. The aggregation results show that the FIB size can be reduced to 1/3 of the original table size, at most. According to [35], even with the state-of-the-art FIB lookup algorithm, called Tree Bit Map [37], the actual memory-saving is half of the original FIB memory and reducing more than this seems impossible. Therefore, instead of FIB aggregation, our work focuses on FIB caching.

FIB aggregation can reduce the FIB size by aggregating a large FIB table into a smaller one with the same forwarding results. 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 [38] 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 but 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.

Forwarding table compression [39] can further increase the space saving benefits of our FIB caching algorithm. By compressing the most popular prefixes, the FIB's memory footprint can be reduced even more.

Liu proposed Routing Prefix Caching for network processors [12] 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. Akhbarizadeh and Nourani proposed RRC-ME [10]. 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. Kim et al. proposed route caching using flat and uniform prefixes of 24 bits long [11]. It can reach 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.

Katta et al. proposed a CacheFlow system [40] to install popular fine-grained forwarding policies in the small TCAM, while relying on software to deal with the traffic with unpopular rules. In their work, the authors have to handle rules that partially overlap, rather than the only

"containment" relationships handled in our proposed work.

7. Conclusion

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 3.5 K prefixes to reach a hit ratio of 99.25% and 20 K 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. Also, our scheme includes a set of efficient algorithms to process both FIB and cache update events while preventing the cache-hiding problem. In addition to simulation, we experimented a new cache timeout algorithm based on the proposed scheme using the OpenFlow protocol and the experimentation closely approximates the simulation results. Our future work includes: (1) investigate other potential cache initialization schemes; (2) design more effective timeout algorithms to reach higher hitting ratio for the OpenFlow implementation; and (3) customize the OpenFlow protocol to include switch-defined cache replacement methods and flow entry requests based on longest idle time or lowest packet count.

Acknowledgments

We thank Christos Papadopoulos, Daniel Massey, and Kaustubh Gadkari from Colorado State University for sharing their traffic trace; Syed Obaid Amin for his contributions to the previous version of this paper; and the anonymous reviewers for their feedback.

Appendix A. Timeout algorithms

Algorithm 1. On prefix installation

```
1: if cache miss then
2:   Lookup the corresponding prefix
3:   if prefix has been installed previously then
4:     Stop prefix.timer
5:     if prefix.removals = 1 then
6:       prefix.timeout ← 1100 s
7:     else if prefix.removals greater than 1 then
8:       prefix.timeout ← prefix.prev_timeout ∗ 2
9:     else
10:      prefix.timeout ← 43 s
11:    end if
12:  else
13:    prefix.timeout ← 43 s
14:  end if
15:  prefix.in_cache ← TRUE
16: endif
```

Algorithm 2. On removal from cache

```
1: Lookup the corresponding prefix
2: prefix.in_cache ← FALSE
3: Increment prefix.removals
4: Start prefix.timer
```

Algorithm 3. On time advance

```
1: for each prefix entry do
2:   if prefix.in_cache = FALSE then
3:     if prefix.timer has reached 0 s then
4:       prefix.removals = 0
5:     end if
6:   end if
7: end for
```

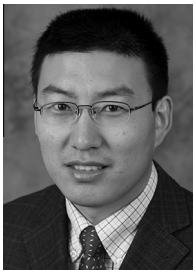
References

[1] Y. Liu, S.O. Amin, L. Wang, Efficient fib caching using minimal non-overlapping prefixes, ACM SIGCOMM Comput. Commun. Rev. (2013) 14–21.
[2] D. Meyer, L. Zhang, K. Fall, Report from the IAB Workshop on Routing and Addressing, RFC 4984. <http://www.ietf.org/rfc/rfc4984.txt>.
[3] RRG, IRTF Routing Research Group, 2008. <http://www.irtf.org/charter?gtype=rg&group=rrg>.
[4] T. Li, Preliminary Recommendation for a Routing Architecture, rFC 6115, 2011.
[5] G.R. Operations, IETF Global Routing Operations (GROW), 2014. <http://www.ietf.org/dyn/wg/charter/grow-charter.html>.
[6] R. Steenbergen, An Inconvenient Prefix: Is Routing Table Pollution Causing DataCenter Warming? 2010. <http://www.nanog.org/meetings/nanog50/abstracts.php?pt=all&nm=nanog50&printvs=1>.
[7] K. Fall, P.B. Godfrey, Routing tables: is smaller really much better?, in: Proceeds of HOTNETS, 2009.
[8] V. Fuller, Scaling Issues with Routing + Multihoming, 1996. <http://www.vaf.net/vaf/apricot-plenary.pdf>.
[9] X. Zhao, D.J. Pacella, J. Schiller, Routing scalability: an operator's view, IEEE J. Sel. Areas Commun. 28 (8) (2010) 1262–1270, http://dx.doi.org/10.1109/JSAC.2010.101004.
[10] M.J. Akhbarizadeh, M. Nourani, Efficient prefix cache for network processors, in: Proceedings of the High Performance Interconnects, 2004.
[11] C. Kim, M. Caesar, A. Gerber, 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.
[12] H. Liu, Routing prefix caching in network processor design, in: Tenth International Conference on Computer Communications and Networks, 2001.
[13] W. Zhang, J. Bi, J. Wu, B. Zhang, Catching popular prefixes at as border routers with a prediction based method, Comput. Netw. 56 (4) (2012) 1486–1502, http://dx.doi.org/10.1016/j.comnet.2012.01.003.
[14] K. Gadkari, D. Massey, 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.
[15] J. Rexford, J. Wang, Z. Xiao, Y. Zhang, BGP routing stability of popular destinations, in: Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurement, IMW '02, 2002.
[16] D.R. Morrison, PATRICIA-practical algorithm to retrieve information coded in alphanumeric, J. ACM 15 (4) (1968) 514–534.
[17] D. Shah, P. Gupta, Fast updating algorithms for tcams, IEEE Micro 21 (1) (2001) 36–47, http://dx.doi.org/10.1109/40.903060.
[18] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, Openflow: enabling innovation in

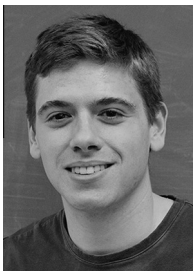
campus networks, SIGCOMM Comput. Commun. Rev. 38 (2) (2008) 69–74. <http://archive.openflow.org/documents/openflow-wp-latest.pdf>.
[19] Openflow Components. <http://archive.openflow.org/wp/openflow-components>.
[20] Open Networking Foundation, Openflow Switch Specification: Version 1.4.0 (Wire Protocol 0x05), October 2013. <https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf>.
[21] Hp 3800 Switch Series. <http://h17007.www1.hp.com/us/en/networking/products/switches/HP_3800_Switch_Series>.
[22] L. Zhao, R. Iyer, S. Makineni, L. Bhuyan, Anatomy and performance of ssl processing, in: 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), vol. 0, 2005, pp. 197–206. <http://doi.ieeecomputersociety.org/10.1109/ISPASS.2005.1430574>.
[23] A.R. Curtis, J.C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, S. Banerjee, Devoflow: Scaling flow management for high-performance networks, SIGCOMM Comput. Commun. Rev. 41 (4) (2011) 254–265, http://dx.doi.org/10.1145/2043164.2018466. <http://doi.acm.org/10.1145/2043164.2018466>.
[24] A. Vishnoi, R. Poddar, V. Mann, S. Bhattacharya, Effective switch memory man-agement in openflow networks, in: Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems, DEBS '14, ACM, New York, NY, USA, 2014, pp. 177–188. http://dx.doi.org/10.1145/2611286.2611301. http://doi.acm.org/10.1145/2611286.2611301.
[25] Mininet: An Instant Virtual Network on Your Laptop (or Other pc). <http://mininet.org>.
[26] Open Vswitch. <http://openvswitch.org>.
[27] About Pox. <http://www.noxrepo.org/pox/about-pox/>.
[28] Open Networking Foundation, Openflow Switch Specification: Version 1.0.0 (Wire Protocol 0x01), December 2009. <https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf>.
[29] CAIDA, Caida Data Trace. <http://www.caida.org/data/monitors/passive-equinix-sanjose.xml>.
[30] Advanced Network Technology Center at University of Oregon, The RouteViews Project, 2005. <http://www.routeviews.org/>.
[31] R. Draves, C. King, S. Venkatachary, B.D. Zill, Constructing optimal IP routing tables, in: Proc. IEEE INFOCOM, 1999.
[32] W. Herrin, Opportunistic Topological Aggregation in the RIB-FIB Calculation? 2008. <http://www.ops.ietf.org/lists/rrg/2008/threads.html#01880>.
[33] X. Zhao, Y. Liu, L. Wang, B. Zhang, On the aggregatability of router forwarding tables, in: Proc. IEEE INFOCOM, 2010.
[34] Y. Liu, X. Zhao, K. Nam, L. Wang, B. Zhang, Incremental forwarding table aggregation, in: Proc. IEEE Globecom, 2010.
[35] Z.A. Uzmi, M. Nebel, A. Tariq, S. Jawad, R. Chen, A. Shaikh, J. Wang, P. Francis, SMALTA: practical and near-optimal FIB aggregation, in: Proceedings of the Seventh Conference on Emerging Networking EXperiments and Technologies, CoNEXT '11, 2011.
[36] Y. Liu, B. Zhang, L. Wang, Fifa: fast incremental fib aggregation, in: IEEE INFOCOM, 2013.
[37] W. Eatherton, G. Varghese, Z. Dittia, Tree bitmap: hardware/software ip lookups with incremental updates, SIGCOMM Comput. Commun. Rev. 34 (2) (2004) 97–122, http://dx.doi.org/10.1145/997150.997160. <http://doi.acm.org/10.1145/997150.997160>.
[38] H. Ballani, P. Francis, C. Tuan, J. Wang, Making routers last longer with ViAggre, in: NSDI, 2009.
[39] G. Rétvári, J. Tapolcai, A. Kőrösi, A. Majdán, Z. Heszberger, Compressing ip forwarding tables: towards entropy bounds and beyond, Proc. ACM SIGCOMM (2013) 111–122.
[40] N. Katta, O. Alipourfard, J. Rexford, D. Walker, Rule-Caching Algorithms for Software-Defined Networks.

Further reading

[41] OpenFlow Tutorial. http://www.openflow.org/wk/index.php/OpenFlow_Tutorial.
[42] Mininet: rapid prototyping for software defined networks. http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet.
[43] Wireshark: the world's foremost network protocol analyzer. http://www.wireshark.org/.
[44] D. Abrahams. Boost.Python. http://www.boost.org/doc/libs/1_53_0/libs/python/doc/, 2003.



Yaoqing Liu is an assistant professor of Computer Science at Clarkson University, USA. He received his bachelor of Computer Science degree from Dalian Maritime University in China. He received both his master's degree and Ph.D. in Computer Science (networking) from the University of Memphis. His research interests include networked systems (routing, security, algorithm, measurement and protocol), software defined networking, future Internet architecture, and named data networking.



Vince Lehman is an undergraduate research assistant at the University of Memphis Networking Research Lab. He will graduate with a bachelor's in Computer Science from the University of Memphis in December 2014. His research interests include software defined networking, performance in networked systems, and named data networking.

Lan Wang is an associate professor in the Computer Science Department at the University of Memphis. She holds a B.S. degree (1997) in Computer Science from Peking University, China and a Ph.D. degree (2004) in Computer Science from University of California, Los Angeles. She is an IEEE senior member and received an Early Career Research Award (ECRA) from the College of Arts and Sciences at the University of Memphis. Her research interests include Internet architecture, Internet routing, network security, network performance measurement, and sensor networks.