An Ultra-fast Universal Incremental Update Algorithm for Trie-based Routing Lookup

Tong Yang, Zhian Mi, Ruian Duan, Xiaoyu Guo, Jianyuan Lu, Shenjiang Zhang, Xianda Sun and Bin Liu
Department of Computer Science and Technology, Tsinghua University, Beijing, China

***Abstract*—With the rapid growth of the Internet, the update messages in backbone routers become more and more frequent due to the ever-increasing dynamic changes on network topologies and new emerging functionalities of the Internet. In addition, update messages often come as a burst. Update action interrupts the packet lookup operation in the router's data plane, thus inefficient incremental update algorithm slows down IP lookup speed, and potentially badly degrades the system performance during bursty updates.**

Among trie-based routing lookup algorithms, binary trie[1] has the best update complexity *O(W)* (*W is the maximum depth of the trie*), but exhibits slow lookup speed, failing to be competent for forwarding tens of gigabit-per-second traffic in backbone routers. Therefore, various improved routing lookup algorithms are proposed to pursue high speed based on binary trie, but sacrificing the performance of incremental update.

To minimize the interruption time that update operation incurs, we propose *Blind Spot* (BS) algorithm by picking out those updating nodes which would have produced domino effect[2], achieving an update complexity of *O(lookup+h)*[3], meanwhile keeping the lookup speed almost unchanged. Blind Spot algorithm is a universal methodology, which is applicable to all the trie-based lookup algorithms. To evaluate the performance of BS algorithm, we applied it to Lulea [1] and LC-trie [2] algorithms as two representatives. Extensive experimental results show that both Lulea+BS and LC+BS algorithms achieve a much faster update speed than binary trie, while keeping the same lookup speed as the original Lulea and LC-trie algorithms.

I. INTRODUCTION

With the rapid increase of Internet traffic, backbone links with tens of gigabit-per-second line rate are deployed, which implies the backbone routers must be able to forward up to tens of millions of packets per second on each port. Consequently, routing lookup, which uses the destination IP address to determine the next-hop for each packet based on the exponential growing routing tables according to the Longest Matching Rule (LPM), becomes a major performance bottleneck in the forwarding plane of the backbone routers.

Routing lookup solutions can be classified into two categories: hardware-based and software-based approaches. Hardware-based methods need special hardware assistance, such as TCAM or GPU, suffering from high hardware cost and high power consumption. Software-based approaches operate on general memory (such as SDRAM), thus can concentrate on the storage and lookup optimization by algorithmic improvements without extra cost. Therefore, software-based approaches, especially trie-based solutions are focused in this paper. The pioneer work of routing lookup is the binary trie [3], owing to its simplicity and ease of use. However, in the worst case, it needs 32 memory accesses for a single lookup, and this makes naïve binary trie way not applicable to high speed situations (such as tens of gigabit-per-second link). Hence, various improved approaches are proposed to increase the lookup speed of binary trie, such as Path-compressed trie, K-stride Multibit trie, LC-trie, Lulea, Full compression, Binary search on prefix lengths, Binary range search, Multiway range search and Multiway range trees [3] [4] (see Table I). All these algorithms trim the trie into a 'trimmed trie', which is beneficial to routing lookup unilaterally, but inevitably loses the simplicity of the original binary trie, sacrificing the incremental update performance. Consequently, none of them can reach the update speed of the binary trie, which has an update complexity *O(W)*[4]. A more sophisticated lookup algorithm often goes along with a more complicated update procedure. For example, LC-trie and Lulea algorithms can achieve fast lookup speed and high memory efficiency. They only need three or four memory accesses for one lookup in average, achieving tens of millions search per second by using up-to-date SRAM, which is close to the lookup speed of hardware-based solutions, but can hardly operate incremental update.

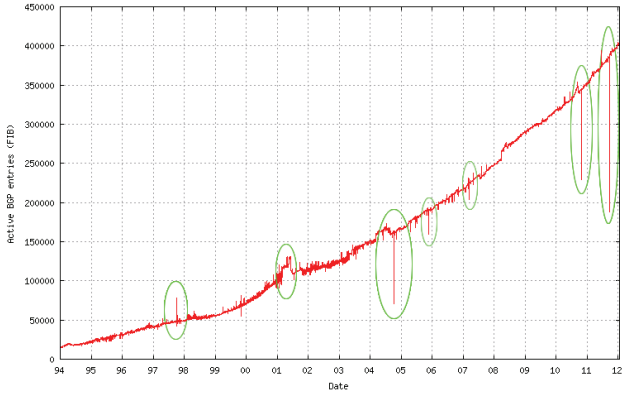


Fig. 1. The size of routing table of AS6447 over the past 19 years.

Making the matter worse, routing updates become more

Authors' Email: {yangtongemail, mzaort}@gmail.com, {dra08, Guoxiaoyu10, lu-jy11, zsj09, sunxd08}@mails.tsinghua.edu.cn, liub@tsinghua.edu.cn.
Supported by NSFC (61202489, 61073171, 60873250), Tsinghua University Initiative Scientific Research Program (20121080068), the Specialized Research Fund for the Doctoral Program of Higher Education of China (20100002110051).

[1]Trie is a tree-like data structure allowing the organization of prefixes on a digital basis by using the bits of prefixes to direct the branching [3].

[2]An update message may incur movements of many prefixes which is the most time-consuming operation during update process. We call this phenomenon 'domino effect' in this paper.

[3]*h* is the maximum depth of Blind Zone, and the meaning of Blind Zone is detailed in Section I.

[4]W is the maximum depth of the trie.

and more frequent. Taking the backbone router AS6447 as an example, the changes of its routing table size over the past years are shown in Figure 1 [5]. Besides the rapid growth of routing table size, this figure also shows the update messages often come as a burst[5]: the curve is toothed, and many bursty updates occur (see the eclipses in Figure 1). Based on our statistical results, the received update messages of the backbone routers reached up to 35K/s in the peak. Similar results are also reported in [5] and [6] – the received update messages reach tens of thousands of prefixes per second in the peak. An update operation will interrupt routing lookup, with a dozen times of per lookup in average, which will suspend the routing lookup operation for a relatively long time with a potential risk of incurring packet loss if the lookup queue goes full during bursty updates. Therefore, the update performance of lookup algorithms has been raised to be an important concern nowadays.

In order to achieve high performance, backbone routers must gracefully handle the update messages, as well as the bursty updates. Then is there a fast incremental update algorithm which can be applied to any trie-based lookup algorithms? This is the issue studied in this paper.

The ideal goal is to propose a novel universal incremental update mechanism, which can operate fast incremental update without degrading the lookup speed of the original algorithms. We propose Blind Spot (BS) algorithm to achieve this goal. Before going into its details, the origin and formation process are introduced.

In our previous work[7][8], we have carried out a series of studies and experiments on real routing tables and update messages, which are downloaded from www.ripe.net [9]. Via our data mining on routing update messages, we discover two significant characteristics of the routing updates:

a) Stabilization Characteristic: Although the routing tables are huge, only a very minute fraction of prefixes are frequently updated. For convenience, 'stable prefixes' are used to represent those prefixes which do NOT change over one day, while 'unstable prefixes' stand for those which update over one day in this paper. Experimental results show that the number of 'stable prefixes' is 86.7% of the whole routing table.

It is easy to understand this characteristic: The routing table was originally designed to be stable, though a dynamic routing protocol is adopted. However, there are indeed some unstable networks, resulting in frequent update messages.

b) Leaves Characteristic: Although the routing table updates frequently, most of the updates occur in the leaf nodes. This characteristic is also easy to understand: the leaf nodes often represent the edge networks, which are usually small and unstable, thus update frequently.

The *rationale* of BS algorithm is to make full use of these two characteristics, pick out those updating nodes which would have incurred domino effect (*if many prefixes must be moved for one update message, we call this domino effect*), and set them as Blind Spots. The work principle of BS algorithm is illustrated in Figure 2. Most of the existing algorithms usually trim the original trie into the 'trimmed trie' which avails to small routing table size and/or fast lookup. With regard to the existing trie-based algorithms, some updates change the structure of the trimmed trie, incur movements of many prefixes and make the update process complicated and inefficient. This kind of domino effect is the most time-consuming operation during update process. The *core idea* of BS algorithm is to eliminate domino effect by setting those updating nodes which would have produced domino effect as Blind Spots, so as to operate fast update.

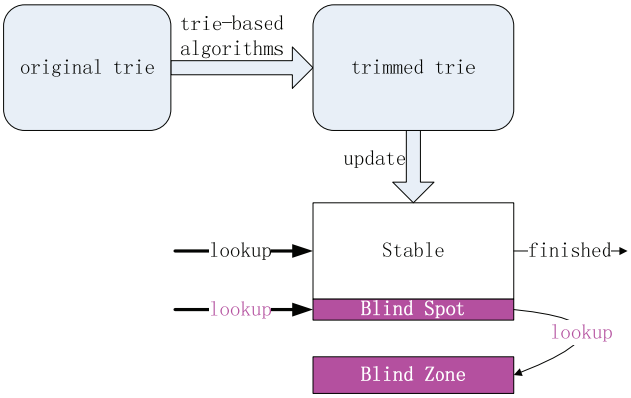


Fig. 2. The work principle of BS algorithm.

BS algorithm classifies the update messages into two categories: 1) updates nodes which do not produce domino effect can be easily handled; 2) updates nodes which would have incurred domino effect are set as Blind Spots, so as to keep the structure of trimmed trie unchanged, meanwhile the updated information is stored in another separate memory zone (we call it Blind Zone in this paper) in the shape of trie branches. In the worst case, BS algorithm needs to locate the Blind Spot, and then updates the corresponding trie branch in the Blind Zone. Suppose the average depth of Blind Zone is *h,* the lookup complexity of the adopted lookup algorithm is *O(lookup)*, then the update complexity of BS algorithm is *O(lookup+h)*. For routing lookup, if the Blind Spot is not hit, the lookup will finish quickly; otherwise, the trimmed trie will be unable to 'see' the next-hop, but it knows where to find the next-hop – the corresponding trie branch in Blind Zone.

It is possible that BS algorithm might exhibit poor performance if there are too many Blind Spots or the Blind Zone is too large or too deep. In fact, the Blind Spots are rare because of the 'stabilization characteristic', and the Blind Zone is also small and shallow because of the 'leaves characteristic'. To evaluate the performance of BS algorithm, we have applied BS algorithm to Lulea scheme[6], named Lulea+BS algorithm. The lookup complexity of Lulea algorithm is *O(W/k)*, and the update complexity of Lulea+BS is *O(W/k+h)* (see Table I). Experimental results show that *h* is no more than 2, and for update, the average memory access count of Lulea+BS (*W/k+h*) is 7.49, while that of binary trie is 22.9. With regard to the lookup speed, experimental re-

[5]Because update messages are very bursty in practice, we use 'bursty update' in this paper to represent those update messages which arrive a lot in a short time.

[6] Lulea algorithm is a representative trie-based approach proposed in [1] , which constructs compact lookup table to achieve fast lookup, but fails to operate incremental update.

sults show that the memory access count of Lulea+BS is almost the same as that of Lulea.

The proposed BS algorithm is a universal fast incremental methodology. For more applications, we choose another representative lookup algorithm – LC-trie algorithm [2]. It combines Level Compression algorithm with Path Compression algorithm [3], exhibiting an excellent lookup performance. However, this combination incurs complicated incremental update procedure. We apply BS algorithm to LC-trie scheme, named LC+BS algorithm, which exhibits both excellent lookup and update performance.

After implementing Lulea and LC-trie algorithms by using C++ language, in order to guarantee the correctness of our implemented algorithms, we finish two corresponding verifying programs[7] to examine the correctness of the implementation of Lulea and LC-trie algorithms.

Specifically, we made the following main contributions:

- Based on the two significant characteristics of routing update messages discovered during massive data mining, we propose Blind Spot algorithm by picking out those updating nodes which would have produced domino effect, enabling a fast updating while almost NOT degrading the lookup throughput. It can be applied to all the trie-based lookup algorithms.
- We make case studies to apply Blind Spot algorithm to Lulea and LC-trie algorithms. The performance of BS algorithm is evaluated by mathematical analyses and extensive experiments: Lulea+BS and LC+BS algorithms achieve a much faster update speed than binary trie while keeping the same lookup speed as the original algorithms.

The remaining parts of this paper are organized as follows. Section II reviews relevant researches related to trie-based routing lookup. Section III presents BS algorithm. Section IV applies BS algorithm to Lulea algorithm, while Section V applies BS algorithm to LC-trie algorithm. Mathematical analyses on BS algorithm's performance are presented in Section VI. Section VII carries out extensive experiments over real data set and trace. Finally, we conclude this conclusion in Section VIII.

II. Related Work

As mentioned above, there are two kinds of routing lookup solutions: hardware-based solutions and software-based solutions. The hardware-based solutions need special hardware support, such as TCAM and GPU. The main advantage of TCAM and GPU is that they can perform parallel lookup, and their main shortcomings lie in the high hardware cost and high power consumption, as reported in [10] and [11].

With the development of VLSI (Very Large Scale Integration) technologies, SRAM devices get faster and faster in speed but cheaper and cheaper in price than ever before, together with increased capacity per unit chip. Thus the improved trie-based lookup algorithms again obtain ISPs' interests given its flexibility and low implementation cost compared with hardware solutions.

The software-based solutions include bloom filter-based solutions [12][13] and trie-based solutions. Bloom filter cannot incrementally deal with withdrawal messages, thus counting bloom filter is introduced to fix it. However, besides the false positive for general bloom filters, counting bloom filter also suffers from false negative. Therefore, existing bloom filter-based algorithms suffer from either incorrect lookup or incremental update disability. Comparatively, some sophisticated trie-based algorithms can approach the lookup speed of hardware solutions, so this paper chooses trie-based algorithms as the basis targeting at improving their update performances while keeping their searching throughput.

For trie-based algorithms, there is an inherent conflict between fast lookup and incremental update, because fast lookup algorithms require sophisticated design of trie structure, accordingly bringing challenges and difficulties to incremental update. The typical trie-based algorithms include: Path-compressed trie, K-stride Multibit trie, LC-trie, Lulea, Full compression, Binary search on prefix lengths, Binary range search, Multiway range search, and Multiway range trees [3] [4] (see Table I). It can be concluded that binary trie achieves the fastest update performance, but a poor lookup speed, while other algorithms are seeking a proper trade-off between lookup and update performance.

TABLE I[8]
TYPICAL TRIE-BASED LOOKUP ALGORITHMS

Classical lookup algorithms	Lookup complexity	Update complexity	Predicted Update complexity using BS
Binary trie	$O(W)$	$O(W)$	-
Path-compressed tries	$O(W)$	$O(W)$	-
K-stride multibit trie	$O(W/k)$	$O(W/k+2^k)$	$O(W/k+h)$
LC-trie	***O(W/k)***	**-**	***O(W/k+h)***
Lulea	***O(W/k)***	**-**	***O(W/k+h)***
Full expansion/compression	*3*	-	*3+h*
Binary search on prefix lengths	$O(\log_2 W)$	$O(N\log_2 W)$	$O(\log_2 W+h)$
Binary range search	$O(\log_2 N)$	$O(N)$	$O(\log_2 N+h)$
Multiway range search	$O(\log_k N)$	$O(N)$	$O(\log_k N+h)$
Multiway range trees	$O(\log_k N)$	$O(k\log_k N)$	$O(\log_k N+h)$
BST(binary prefix tree)	$O(W)$	$O(W)$	-
Priority Trie	$O(W)$	$O(W)$	-
BSR (Binary search on range)	$O(\log_{k+1} N)$,	$O(N)$	$O(\log_{k+1} N)+h$

Is the trade-off between lookup and update indispensable? Can't the update complexity $O(W)$ of binary trie be exceeded? The answer is negative. This paper presents a universal update algorithm, which makes it possible that the faster the lookup is, the faster the incremental update will be. The predicted update complexity of using BS algorithm is presented in the last column of Table I. Given the space limitation, only two of them: Lulea and LC-trie algorithms, are chosen for our case studies, because Lulea algorithm fails to support

[7] Take the verifying program of Lulea as an example: for the same IP address, the program compares the next-hop of Lulea and binary trie. If and only if all the comparisons are equal by traversing IPv4's 32-bit space, the implementation of Lulea algorithm is correct; otherwise, the program stops and tells the prefix and the wrong next-hops. One whole comparison takes about 20 minutes for a 400K routing table. Only in this way can we guarantee the ultimate correctness of Lulea and LC-trie.

[8] For K-stride Multibit tries and LC-trie, k is the maximum stride size in bit in the multibit trie. For Lulea, k is the number of table degree of the new constructed table. For Multiway range search and Multiway range trees, k means the number of branches of each node. For BSR, k is the number of search keys in a search node.

incremental update and the update of LC-trie algorithm is difficult and complicated. In fact, BS algorithm is a universal methodology and can be applied to any of the trie-based algorithm. To the best of our knowledge, this is the first effort on a universal fast incremental update mechanism for trie-based routing lookup algorithms.

III. BLIND SPOT ALGORITHM

A. Terms and Definitions

The following terms will be used in this paper, so their definitions are provided in Table II.

TABLE II
TERMS AND DEFINITIONS

Terms	Definitions
FIB	Forwarding Information Base
BS	Blind Spot
BZ	Blind Zone
W	The maximum depth of trie
O(lookup)	The routing lookup complexity of an algorithm
Oldport	The next-hop of a prefix in FIB before trimmed
Newport	The next-hop of a prefix in FIB after trimmed

B. Blind Spot Algorithm

As mentioned above, routing updates become more and more frequent. Consequently, some sophisticated trie-based algorithms fail to be well applied to real-world routers due to their complicated update, such as Lulea and LC-trie algorithms. In [1], the authors supposed the routing table is stable, and the update messages are very rare. In [2], the authors didn't mention the corresponding incremental update algorithm.

Update operation interrupts routing lookup, thus fast incremental update algorithm will contribute to high lookup speed. As shown in Table I, we can see that among all the existing trie-based algorithms, binary trie achieves the best update complexity *O(W)*. Can't *O(W)* be exceeded? The answer is negative, and the following gives the reason.

1) The rationale of Blind Spot algorithm

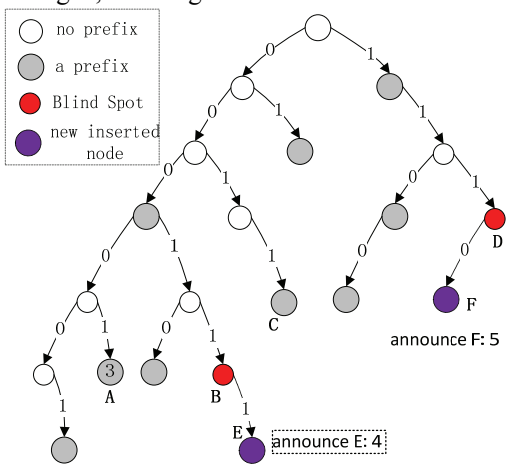
A Blind Spot, also known as a scotoma, is an obscuration of the visual field. Just imagine, a routing table is like an eye, and each prefix is like a visual spot. When a lookup request arrives, the longest matched prefix will be found in the routing table, just as that this eye (the routing table) sees the visual spot (the matched prefix). In order to achieve fast lookup, existing algorithms usually trim the trie, *i.e.*, arrange the prefixes (the visual spot) in a compact structure, which is in favor of the memory-efficiency of the routing lookup algorithms, but inevitably bringing difficulties to incremental update.

The rationale of Blind Spot algorithm is to keep the compact structure (like the visual field) unchanged during update process to avoid domino effect: if the updates will not change the trimmed structure, *i.e.*, not incur domino effect, just making simple update operations; otherwise, set the affected nodes (like the infected visual spot) to Blind Spots, and record the relevant update information in Blind Zone. Actually, the Blind Spots are very few because of 'stabilization characteristic', and the Blind Zone is small and shallow because of the 'leaves characteristic'.

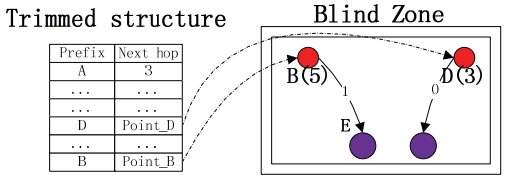
BS algorithm is a novel universal incremental update scheme, which can operate fast incremental update with a complexity *O(lookup+h)*, while keeping its lookup speed almost unchanged. The details of Blind Spot algorithm are elaborated below.

2) The update mechanism of Blind Spot algorithm

Specifically, when an update message arrives, BS algorithm will judge whether it will change the trimmed structure: if not, just update the trimmed trie quickly (this is convenient and fast); otherwise, figure out and set those affected prefixes as Blind Spots to keep the structure of the trimmed trie unchanged, avoiding domino effect.



(a) The appearance of Blind Spot during updates.



(b) The appearance of Blind Zone during updates.

Fig. 3. The data structure organization of BS algorithm.

In order to make a clearer picture of Blind Spot algorithm, an example is given in Figure 3. Here the 'trimmed trie' is the trie without node E and F. For example, two update messages arrive: 'announce 000111*:4' and 'announce 1110*:5'. They mean inserting a node E with a next-hop 4 as node B's right child and inserting a node F with a next-hop 5 as node D's left child. In order to keep the structure of the trimmed trie unchanged, Blind Spot algorithm sets node B as a Blind Spot, and set node B's next-hop to Point_B which points to the blind trie (trie branch) – B(5), as shown in Figure 3. In the same way, node D is set as a Blind Spot, and the corresponding blind trie D(3) is also built. In practice, the Blind Zone does not need to be built, because FIB is usually stored by a trie in the slow memory (such as DRAM), and Blind Spot algorithm just needs to set Point_B to the sub-trie rooted at B. In this way, Blind Zone does not need additional memory, although Blind Zone is very small according to our experimental results. It is likewise definitely feasible that Blind Zone is stored independently, given its size is small.

3) The lookup procedure of Blind Spot algorithm

Then how to lookup the routing table with Blind Spots? If a lookup doesn't hit Blind Spot, it will finish quickly. Otherwise, a second lookup is needed – those lookups which hit the Blind Spot will access the Blind Zone. To be clearer,

two examples are given below.

As shown in Figure 3, for example, when looking up a prefix 0110*, no Blind Spot node is accessed, the lookup will finish promptly, and the lookup speed is equal to that of the adopted routing table algorithm. For an example in the worst case (see Figure 3), given an prefix 00011100*, the Blind Spot node 'B' is matched, at this moment, the adopted lookup algorithm cannot 'see' the next-hop, but knows where to find it – Point_B, which points to the small blind trie (trie branch) B(5) in the Blind Zone. Then a second lookup happens, the prefix 00011*100** is looked-up in the blind trie B(5), where '5' indicates the level of this blind trie is 5, thus the lookup only needs to start from the sixth bit, in other words, only *100** is looked up in the blind trie B(5). As a result, the purple node E is matched according to LPM, and then the next-hop of 00011100* is node E's next-hop. Only in this case, a second lookup is needed. Nonetheless, because of the 'leaves characteristic', the average depth of the Blind Zone is around 2 (Luea) or 4 (LC-trie), according to our experimental results. In other words, for routing lookup, only 2 or 4 additional memory accesses are required in the worst case.

4) Periodical defragment

The Blind Spots are like the fragments of the disk. After a long time, the Blind Spot might be many, the routing table needs to be reconstructed upon the concrete lookup algorithm. But this procedure will not affect the lookup much.

IV. Application of BS Algorithm on Lulea Algorithm

BS algorithm is a universal methodology, representing a kind of update mechanism rather than a specific approach. It can be applied to all the trie-based lookup algorithms. In this section, Lulea algorithm is chosen for performance evaluation of BS algorithm. The algorithm introduces a new data structure called bitmap for low storage requirement and fast lookup, yielding dramatically different characteristics from trie. The data structure is highly interrelated, consequently Lulea algorithm is hard to make incremental update. As aforementioned, the binary trie updates incrementally faster than all other existing lookup algorithms. This section will present an inspiring result: Lulea+BS algorithm exhibits an ultra-fast update speed, which is superior to binary trie update.

To achieve fast routing lookup speed and small memory requirement, Lulea algorithm builds three tables based on a trimmed trie – 'complete trie'[9], by using Leaf-pushing algorithm [14].

It is noteworthy to mention that only those updates which interrupt routing lookup have impacts on system performance. In order to reduce the interruption time of routing lookup as much as possible, the update messages are preprocessed, *i.e.*, being translated into different update messages with two parts: one interrupts routing lookup, and the other does not. The preprocessing operation does not interrupt lookup, because they access the raw trie which is stored in slow memory, such as DRAM, while the fast memory (such as SRAM) stores routing table which is used to perform routing lookup.

1) Preprocessing operations

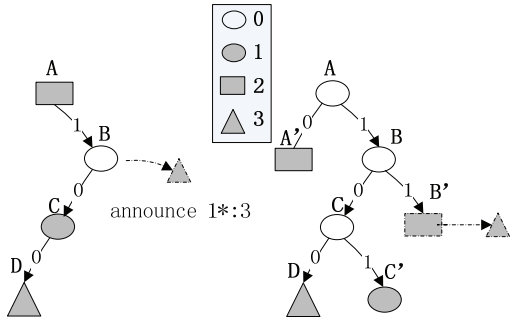
One update message could trigger more than one change on the trimmed trie structure by routing lookup algorithm, but this process can be completed without interrupting routing lookup, and we call it preprocessing or translation. Then how does translation work? The scheme is divided into two situations:

a) Leaf nodes updates

For these updates, *one update message merely affects one node*. These updates are easy to handle – just update the next-hop of the corresponding node.

b) Internal nodes update

The internal nodes updates should be handled carefully, because one update could trigger the change of several nodes even a sub-trie. An example is illustrated in Figure 4. Figure 4(a) is the original trie, and Figure 4(b) is the trimmed trie by utilizing Leaf-pushing. The shape of the node is used to represent its next-hop: solid ellipse, solid rectangle, and solid triangle represent the Newport 1, 2 and 3, respectively, while the next-hop of hollow node is NULL. In this example, the update message is 'announce 1*:3', the 'shadow' of node B should be found and updated to keep the structure of 'complete trie' unchanged. It is obvious that node B' is the shadow of node B (there may be more in practice), thus the next-hop of node B' is changed to 3 (the shape of B' is changed from rectangle to triangle (see Figure 4(b)). This is the preprocessing process of internal nodes.



a) The original trie b) the trimmed trie using Leaf-pushing

Fig. 4. An example when an update occurs in the internal nodes using Leaf-pushing.

2) Update operations

After the preprocessing operations are accomplished, each original update message is translated into one or several nodes update messages, which are divided into two categories:

a) Internal and leaf nodes update: The first category is the update of the internal nodes and the leaf nodes, which does not break the structure of 'complete binary trie'. The leaf nodes update are easy to handle. For the internal nodes update, the complexity of this part is the same as lookup complexity, *i.e.*, *O(m(w/k))*, where '*m*' indicates the number of nodes update translated by one original update. Only the update of internal nodes may incur this situation. According to our experimental results, m is around 2 in average, and the internal nodes updates are only 13.3% of the whole

[9]A complete trie in this paper means that every node in this trie either has two child nodes, or has no node.

updates. Therefore, this part of update does not affect the overall update performance much, so its complexity can be regarded as *O(w/k)*.

b) Leaf Insertion: The second category is the operation of inserting a new node as the leaf nodes' offspring, which would have broken the structure of 'complete trie', incurring domino effect of the data structure of Lulea. To avoid domino effect, the structure of 'complete trie' should be kept unchanged, thus BS algorithm first sets the nearest ancestor node which belongs to the 'complete trie' as Blind Spot. The complexity of this step is the same as the lookup complexity *O(W/k)*. The subsequent step of BS algorithm is to update the corresponding trie in Blind Zone. Because the average depth of Blind Zone is 2, the complexity of this step is *O(2)*. In sum, the complexity of the second category is *O(W/k+2)*.

According to the analyses of 1) and 2), it can be concluded that the update complexity of Blind Zone algorithm is *O(W/k+2)*. To extend this conclusion, any trie-based lookup scheme combined with BS algorithm can achieve an update complexity *O(lookup+h)*, while *h* is the average depth of Blind Zone. This is an exciting result.

V. Application of BS Algorithm on LC-trie Algorithm

In [2], Stefan Nilsson et al. proposed the classical LC-trie – a trie structure, which combines Path Compression and Level Compression, so as to build efficient, compact, and easily searchable implementation of a routing table. Because the update of both level compression and path compression is not simple, the update of their combination – LC-trie is rather complex. We apply BS algorithm to LC-trie algorithm, named LC+BS algorithm. By picking out those updates which are most time-consuming, the update process of LC-trie becomes fast. The update scheme of LC+BS can be classified into two categories:

A. The Update of Level Compression

Level compression works only in a complete sub-trie, thus the update scheme of LC+BS is divided into the following situations:

1) Inside the complete sub-trie

The nodes inside the complete sub-trie are unaffected by Level Compression. For this kind of update messages, in order to maintain the trie structure, the update algorithm is divided into two steps:

Step 1: Update the changed node. This step is straightforward, just set the next-hop of the changed node to the new next-hop.

Step 2: Update the affected nodes. The affected nodes refer to those nodes which are in the sub-trie with NULL Oldport and directly connected with the updating node. The update mechanism of this step is classified into three cases:

a) Insertion operation: Just change the Newport of the affected nodes to the insertion port;

b) Changing operation: Just change the Newport of the affected nodes to the changing port;

c) Deletion operation: Just change the Newport of the affected nodes to 0;

During the above three steps, the Oldport of the affected nodes are maintained by LC+BS algorithm.

2) The leaf nodes of the complete sub-trie

Note that the leaf nodes of the complete sub-trie are not really leaf nodes, but the nodes which are at the bottom of the complete sub-trie.

In this situation, the update process is convenient: 1) for the insertion and changing operation, just set the next-hop of the changed nodes to update port; 2) for the deletion operation, just set the Newport and Oldport to 0.

B. The Update of Path Compression

1) Inside the path compressed sub-trie

For the updates which occur inside the path compressed sub-trie, if the sub-trie is changed back, a lot of operations and overhead are needed. In this case, LC+BS just:

a) Set the affected nodes to Blind Spots. This step is fast enough with the same complexity as that of LC-trie *O(W/k)*.

b) Put the original sub-trie to Blind Zone. The original sub-trie can be restored according to the Path Compression sub-trie, and can also be obtained from the original trie. Although this step costs some time and memory, it operates in Blind Zone, which is rarely accessed by lookup, thereby almost not affecting routing lookup.

2) The leaf nodes of the path compressed sub-trie

The leaf nodes have the similar meaning with the above leaf nodes of the Level Compression, and their update schemes are the same.

3) The under leaf nodes of the complete sub-trie

This is easy to deal with – just insert a new node with the given next-hop.

C. The Nodes Unaffected by Level and Path Compression

For these nodes unaffected by either Level Compression or Path Compression, the update scheme is natural: just insert, change or delete the corresponding node.

VI. Mathematical Analyses

Before going to experiments, the performance of BS algorithm is evaluated by mathematical method in this section. There are two main metrics of BS algorithm: the effect of lookup speed using BS algorithm and the update complexity of BS algorithm.

A. The Effect of Lookup Speed Using BS Algorithm

Suppose the size of the original routing table is *M*, the prefix count of the trimmed trie is *N*, the count of Blind Spots is *B*, and the average depth of the Blind Zone is *h*. Therefore, the size of Blind Zone is *B*h*.

Intuitively, the access probability of Blind Spots is below the average value, because most Internet traffic is apt to go through the huge and stable networks rather than the acroteric and unstable networks which are the main source of Blind Spots. This intuitive judgment is in conformity with the subsequent experimental results. In order to make arithmetical analyses close to reality, we suppose the access probability of Blind Spots is equal to the average *(B/N)*.

Suppose the traffic is V bps, and each packet is at its minimum size of 40 bytes, thus a router should accomplish

V/8/40 IP lookups per second.

Suppose the average count of memory access of each lookup is C, thus the memory access count that the original algorithm requires is:

$$C_o = \frac{\frac{V}{8}}{40} * C = \frac{1}{320}VC$$

After applying BS algorithm, the memory access count is:

$$C_{BS} < \frac{\frac{V}{8}}{40} * C * \left(1 - \frac{B}{N}\right) + \frac{\frac{V}{8}}{40} * (C + h) * \frac{B}{N} = \frac{1}{320}VC + \frac{V}{320}h\frac{B}{N}$$

Therefore, the additional memory access count is

$$\delta = C_{BS} - C_o < \left(\frac{1}{320}VC + \frac{V}{320}h\frac{B}{N}\right) - \frac{1}{320}VC = \frac{V}{320}h\frac{B}{N}$$

The ratio of the additional memory access count to the original memory access count is:

$$\mu = \frac{\delta}{C_o} < \frac{h}{C} * \frac{B}{N}$$

When BS algorithm is applied to Lulea scheme, experimental results show that B/N (the proportion of Blind Spots to the total prefixes) is no more than 0.8%, the average depth of Blind Zone – *h* is no more than 2, and the memory access count – C is 4.56 in average, therefore,

$$\mu < \frac{h}{C} * \frac{B}{N} = \frac{2}{C} * 0.008 = \frac{2}{4.56} * 0.008 = 0.0035 \quad (1)$$

Similarly, when BS algorithm is applied to LC-trie scheme, experimental results show that B/N is no more than 0.004, *h* is around 4 in average, and C is 3.17 in average, therefore,

$$\mu < \frac{h}{C} * \frac{B}{N} = \frac{2}{C} * 0.004 = \frac{2}{3.17} * 0.004 = 0.0025 \quad (2)$$

In conclusion, according to the inequality (1) and (2), after applying BS algorithm, the additional memory access count is bounded to 0.25% and 0.35% of the original memory access count, and this is so small that the effect BS brings can be ignored.

B. The Update Complexity of BS Algorithm

As for the update performance, the update process of BS algorithm is divided into two steps:

1) Step 1: Update in the Trimmed Trie: For an update message, the updating node should be located and subsequently updated in the 'Trimmed trie'. The complexity of this step depends on the concrete lookup algorithm, which is represented by *O(lookup)* in this paper. In the worst case, when Blind Spots are hit, step II is needed.

2) Step II: Update Blind Zone: Because the Blind Zone consists of small trie branches, thus the complexity of this step is *O(h)* (h is the average depth of Blind Zone).

According to (1) and (2), the update complexity of BS algorithm is *O(lookup+h)*. It suggests that the memory access count of BS algorithm is merely *h* more than the adopted lookup algorithm. According to our experimental results of Lulea+BS, the average memory access count of lookup is 4.56, while that of update is 7.49, 7.49-4.57 is a little bigger than *h* (≈2), because one internal node update might be translated into several node updates.

VII. Evaluation of BS Algorithm

A. Experimental Settings

The routing tables are taken from www.ripe.net [9] at RIPE NCC, Amsterdam, which collects default free routing updates from backbone routers. To test the performance of BS algorithm, the routing tables from three routers (RRC00, RRC03, and RRC05) at 08:00 in 2012.01.01 and the corresponding update data from 2012.01.01/08:00 to 2012.01.02/08:00 are downloaded and parsed.

The real traffic used in lookup experiments is from [15]. The traffic at 20:59, 21:09, and 21:19 on 2011.02.17 in Chicago, lasting for one minute, is downloaded and parsed.

Our experiments have been done on a windows XP SP3 machine with Pentium (R) Dual-Core CPU 5500@2.80GHz and 4GB memory.

B. Experiments on the Characteristics of Update Messages

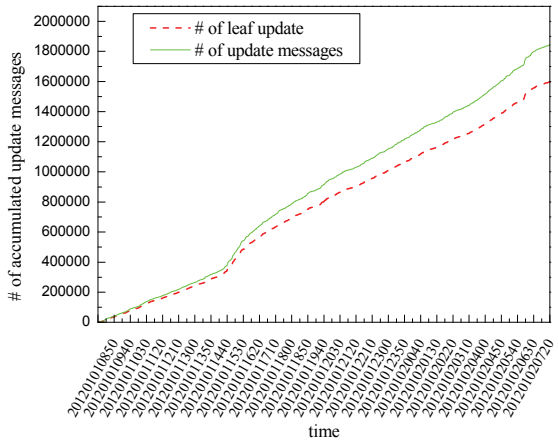


Fig. 5. Comparison between leaf update count and total update count over one day.

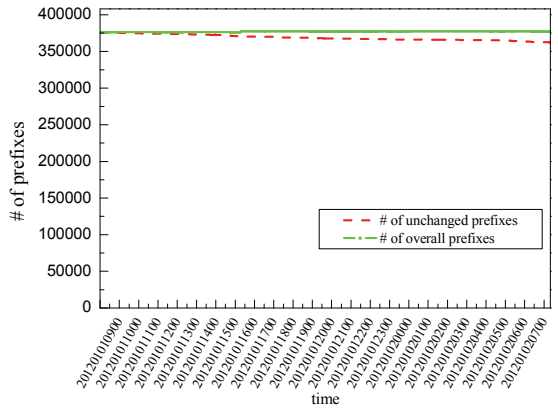


Fig. 6. Comparison between stable node count and total node count over one day.

The x-axes of Figure 5~10 and Figure 14~17 mean the arrival time of the update messages. For instance, '201210231945' represents the time 2012-10-10 23:19:45. Section I states two characteristics of routing updates, which are supported by Figure 5 and Figure 6.

The number of update messages is shown in Figure 5. Results show that the number of update messages reaches 2,000,000 in one day. The higher curve is the number of update messages, while the lower one is the number of leaf node updates. This result supports the 'leaves characteristic':

regardless of the fact that the routing table updates frequently, 86.7% updates occur in the leaf nodes.

Figure 6 shows the number of stable nodes and total nodes over one day. It can be observed that the unchanged nodes decrease slowly, and the number is 362018 over one day, while that of all prefixes is 377051. In other words, only 3.99% nodes update over one day. This result supports the 'Stabilization characteristic': despite the fact that the routing tables are very large, only a very small proportion of prefixes update.

C. Experiments on Lulea+BS Algorithm

As aforementioned, the Blind Spots are rare. In order to verify this conclusion, routing tables from three routers (RRC00, RRC03, and RRC05 [9]) are selected to evaluate the Blind Spot count in Figure 7. This figure shows that the Blind Spot count is only about 3200 after one day. In addition, the BS count grows slower and slower as time goes by.

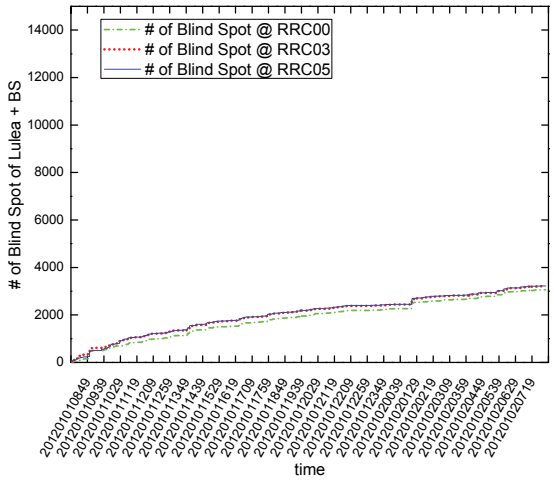


Fig. 7. BS count over one day.

To give a more intuitional result, the ratio of Blind Spot count to all prefixes count is plotted in Figure 8. It can be observed that the ratio increases slowly, and only 0.8% prefixes become Blind Spots after one day.

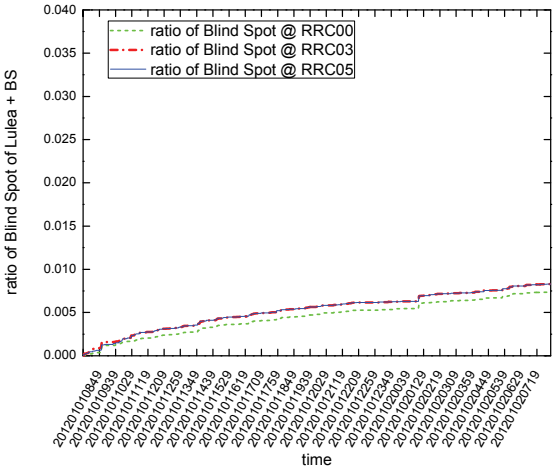


Fig. 8. The proportion of BS count to the total node count over one day.

Because only when one Blind Spot is hit during a lookup process, a second lookup is needed. On average, the probability of the second lookup is only 0.8%. In practice, because the Blind Spots are mostly leaf nodes and unstable nodes, the probability that routing lookup hits Blind Spots will be lower than the average value, and this is supported by Figure 11.

We not only insist that the Blind Spots are rare, but also insist that the Blind Zone (BZ) is small and shallow. In order to verify these conclusions, the experimental results of BZ size are shown in Figure 9, and the evaluation of BZ depth is shown in Figure 10.

As shown in Figure 9, the BZ size, which is defined as the number of all the nodes (including hollow nodes) in the trie of the BZ, increases slowly, and its change is generally in accord with the ratio of BS count. Specifically, the BZ size is only around 10800 after one day.

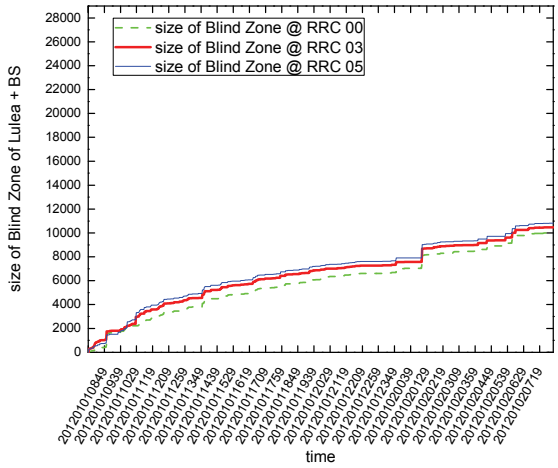


Fig. 9. The size of BZ over one day.

Figure 10 shows the average depth of BZ. Results show that the average depth is relatively steady, and fluctuates below 2. This result suggests that even if a second lookup occurs, only two additional memory accesses will be needed. Then what about the probability that the second lookup occurs? Figure 11 gives the answer.

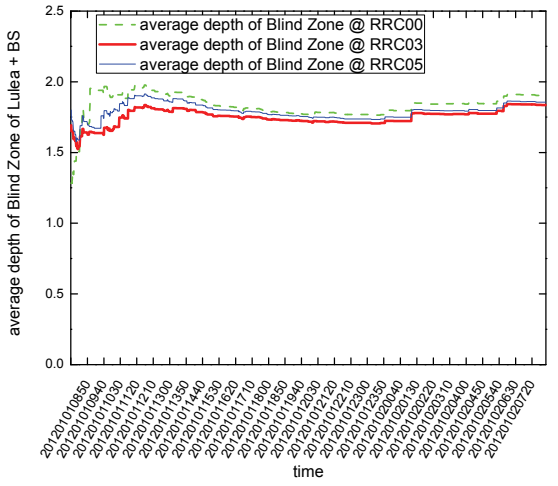


Fig. 10. The average depth of BZ over one day.

In order to make the results more precise and objective, three-interval traffic is chosen to test the proportion of the second lookup. The x-axis of Figure 11 means the seconds in one minute, and the y-axis represents the proportion of the second lookup to the total lookup. The results are similar: the proportion of second lookup to the total lookup is only

0.38%~0.58%. This is in conformity with our previous prediction: the actual proportion of the second lookup is smaller than the average case (0.8%). These results indicate that after applying BS algorithm, the lookup speed of the original lookup scheme is almost unaffected. This will be confirmed again by the following experiments.

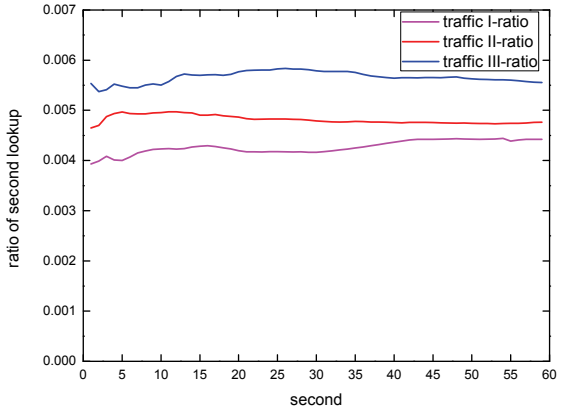


Fig. 11. Proportion of the second lookup to the total lookup over one day.

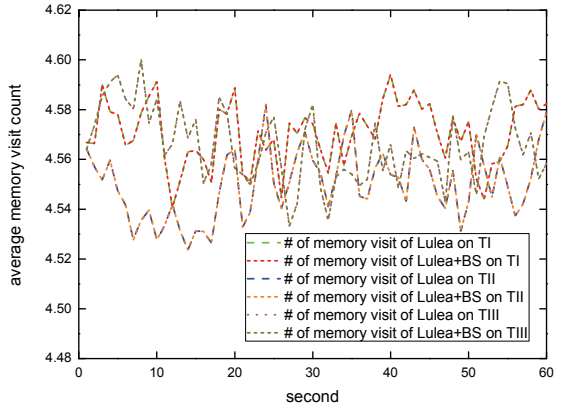


Fig. 12. Comparison of average memory access count between Lulea and Lulea+BS algorithms using three-interval traffic.

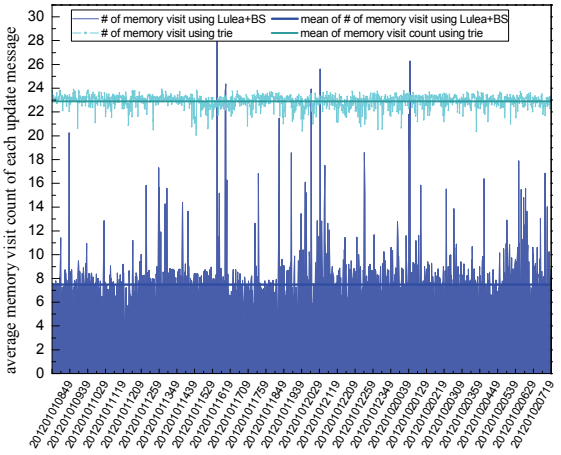


Fig. 13. Comparison of average memory access count between Lulea and Lulea+BS algorithms over one day update.

To evaluate the lookup performance of Lulea+BS algorithm, three-interval traffic is chosen to test the memory access count of Lulea and Lulea+BS algorithms. As shown in Figure 12, the memory access count ranges from 4.5 to 4.6. There are actually six curves, but it seems to be only three, because the memory access count curves of Lulea and Lulea+BS algorithms almost overlap. In other words, because the Blind Spots are rare and the Blind Zone is small and shallow, the lookup performance of Lulea+BS algorithm is almost the same as that of Lulea.

The update performance of Lulea+BS algorithm is evaluated by its memory access count in Figure 13. Results show that the memory access count of Lulea+BS ranges from 1.1 to 28.38 with a mean of 7.49, while that of binary trie is around 22.9. It can be observed that the memory access count of Lulea+BS fluctuates, because one update message might be translated into several nodes update by preprocessing.

D. Experiments on LC+BS Algorithm

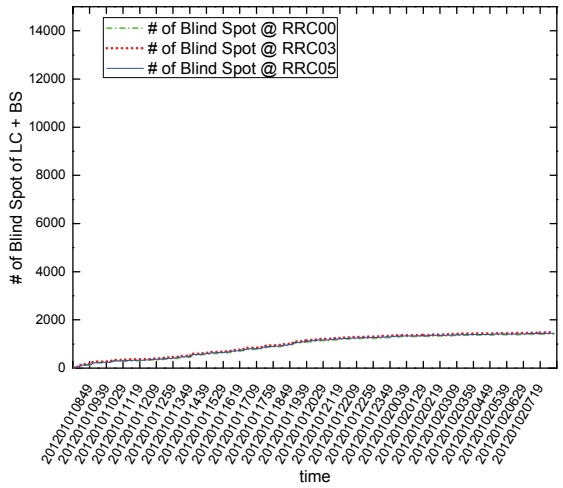


Fig. 14. BS count of LC-trie over one day.

As aforementioned, we also apply BS algorithm to LC-trie scheme, and similar experiments are conducted, and the experimental results are shown in Figure 14~20.

Figure 14 shows the number of Blind Spots is around 1490 after one day, and the ratio of Blind Spots is about 0.39% (see Figure 15), which is smaller than that of Lulea+BS. The size of Blind Zone is around 8600 after one day (see Figure 16), which is also smaller than that of Lulea+BS algorithm.

According to the work principle of BS algorithm, the Blind Spots of LC+BS emerge inside the Path Compression trie, hence the depth of Blind Zone will be bigger than that in Lulea+BS algorithm. The corresponding results are shown in Figure 17, which indicates the average depth of Blind Zone of LC+BS is around 4.

In order to evaluate the effect of BS algorithm on routing lookup, the probability of the second lookup of LC+BS is tested in Figure 18. The results show that the proportion of the second lookup of LC+BS over one day ranges from 0.0324% to 0.0845% with a mean of 0.0576%, which is so small that can be ignored.

The average memory access counts of LC-trie and LC+BS algorithms using three-interval traffic are plotted in Figure 19. Results show that the LC+BS algorithm exhibits almost the same lookup performance as LC-trie algorithm.

With the same lookup performance, LC+BS algorithm achieves a much faster update speed than binary trie: as shown in Figure 20, the memory access count of LC+BS is only 8.22 in average, while that of binary trie is 22.9.

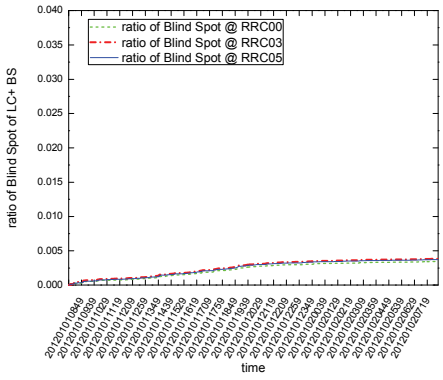


Fig. 15. The proportion of BS count to total node count of LC-trie over one day.

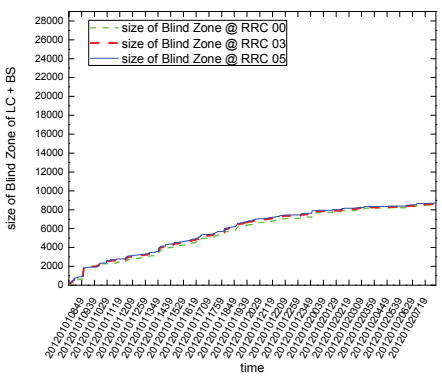


Fig. 16. The size of BZ over one day.

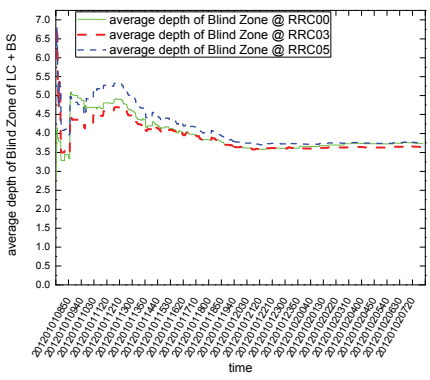


Fig. 17. The average depth of BZ over one day.

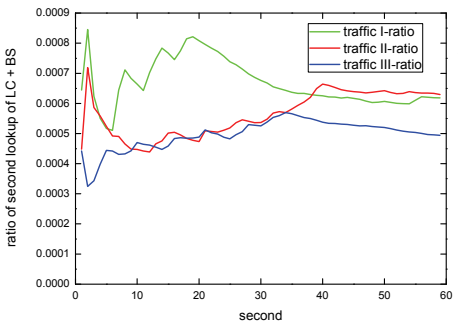


Fig. 18. The proportion of the second lookup to total lookup of LC+BS over one day.

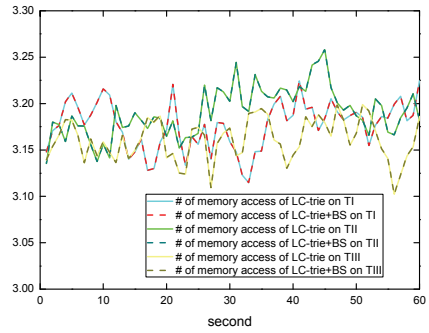


Fig. 19. Comparison of average memory access number between LC-trie and LC+BS using three-interval traffic.

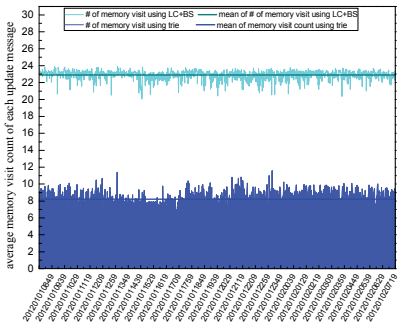


Fig. 20. Comparison of average memory access number between LC-trie and LC+BS over one day update.

VIII. Discussions and Conclusions

With the fast development of the Internet, a major concern of ISPs today is to continue to improve the routing lookup speed. In the pursuit of fast lookup, the incremental update performance usually declines in the existing trie-based lookup algorithms. To cope with the frequent routing updates, especially the bursty updates, a universal update algorithm, named BS algorithm, is proposed. It can achieve an update complexity *O(lookup+h)*, while keeping the original lookup performance almost unchanged. In order to verify this conclusion, BS algorithm is applied to Lulea and LC-trie schemes in this paper. Large-scale experimental results show that the update performance of Lulea+BS and LC+BS algorithms outperforms that of binary trie, without degrading the lookup performance of Lulea and LC-trie algorithms.

References

[1] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, "Small Forwarding Tables for Fast Routing Lookups," Proceedings of ACM SIGCOMM'97, pp. 3-14, September 1997.

[2] S. Nilsson and G. Karlsson "IP-Address Lookup Using LC-Tries," IEEE Journal on Selected Areas in Communications June 1999, Vol. 17, Number 6, pp. 1083-1092.

[3] Miguel Á. Ruiz-Sánchez, Ernst W. Biersack, Walid Dabbous. Survey and Taxonomy of IP Address Lookup Algorithms. Network, IEEE. 2001.

[4] Hyesook Lim, Changhoon Yim, and Earl E. Swartzlander. Priority Tries for IP Address Lookup. IEEE transactions on computers, VOL. 59, NO. 6, JUNE 2010

[5] AS6447 BGP Routing Table Analysis. http://bgp.potaroo.net/as6447/.

[6] Layong Luo, Gaogang Xie, Yingke Xie, Laurent Mathy, Kavé Salamatian. A Hybrid IP Lookup Architecture with Fast Updates. In Proc. IEEE INFOCOM, 2012.

[7] Tong Yang, Ruian Duan, Jianyuan Lu, Shenjiang Zhang, Huichen Dai and Bin Liu. CLUE: Achieving Fast Update over Compressed Table for Parallel Lookup with Reduced Dynamic Redundancy. The 32nd International Conference on Distributed Computing Systems (IEEE ICDCS 2012), Macau, China, June 18-21, 2012.

[8] Tong Yang, Bo Yuan, Shenjiang Zhang, Ting Zhang, Ruian Duan, Yi Wang, and Bin Liu. Approaching Optimal Compression with Fast Update for Large Scale Routing Tables. The 20th International Workshop on Quality of Service (IEEE/ACM IWQoS 2012), Coimbra, Portugal, June 4-5, 2012.

[9] RIPE Network Coordination Centre. http://www.ripe.net/data-tools/stats/ris/ris-raw-data.

[10] Zheng, K., Hu, C., Lu, H., Liu, B. A TCAM-based distributed parallel IP lookup scheme and performance analysis. IEEE/ACM Trans. Netw. 14, 863–875, 2006.

[11] Sangjin Han, Keon Jang, KyoungSoo Park, Sue Moon. PacketShader: a GPU-Accelerated Software Router. In Proc. SIGCOMM, 2010.

[12] Sarang Dharmapurikar, Praveen Krishnamurthy, David E. Taylor. Longest Prefix Matching Using Bloom Filters. In Proc. ACM SIGCOMM, 2003.

[13] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, John Lockwood. Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing. In Proc. ACM SIGCOMM, 2005

[14] V. Srinivasan and G. Varghese, Fast IP lookups using controlled prefix expansion, ACM TOCS, vol. 17, pp. 1-40, Feb. 1999.

[15] The CAIDA Anonymized 2011 Internet Traces - <20110217> Colby Walsworth, Emile Aben, kc claffy, Dan Andersen, http://www.caida.org/data/passive/passive_2011_dataset.xml.