Bitmap-based Prefix Caching for Fast IP Lookup

Jinsoo Kim [1], Myeong-Cheol Ko [1], Junghyun Nam [1] and Junghwan Kim [1]
[1] Department of Computer Engineering, Konkuk University
322 Danwol-dong, Chungju-si, Chungbuk 380-701, Korea
[e-mail: {jinsoo, cheol, jhnam, jhkim}@kku.ac.kr]
*Corresponding author: Junghwan Kim

Received May 24, 2013; revised January 8, 2014; accepted February 27, 2014; published March 31, 2014

Abstract

IP address lookup is very crucial in performance of routers. Several works have been done on prefix caching to enhance the performance of IP address lookup. Since a prefix represents a range of IP addresses, a prefix cache shows better performance than an IP address cache. However, not every prefix is cacheable in itself. In a prefix cache it causes false hit to cache a non-leaf prefix because there is possibly the longer matching prefix in the routing table. Prefix expansion techniques such as complete prefix tree expansion (CPTE) make it possible to cache the non-leaf prefixes as the expanded forms, but it is hard to manage the expanded prefixes. The expanded prefixes sometimes incur a great deal of update overhead in a routing table. We propose a bitmap-based prefix cache (BMCache) to provide low update overhead as well as low cache miss ratio. The proposed scheme does not have any expanded prefixes in the routing table, but it can expand a non-leaf prefix using a bitmap on caching time. The trace-driven simulation shows that BMCache has very low miss ratio in spite of its low update overhead compared to other schemes.

Keywords: IP address lookup, routing table, prefix expansion, prefix cache, bitmap-based caching

This work was supported by Konkuk University.

http://dx.doi.org/10.3837/tiis.2014.03.009

1. Introduction

Today the Internet router needs higher performance than ever before because of rapidly increasing traffic. In the Internet router IP address lookup is one of the major functions to provide fast packet forwarding. The IP address lookup engine has to determine the next hop address and output port by means of a given destination IP address. The advent of CIDR (Classless Interdomain Routing) leads to the longest prefix matching so it requires highly sophisticated searching to reduce lookup time.

Many researches focus on efficient lookup schemes to reduce lookup latency while some works adopt a cache to exploit locality. The average lookup time can be dramatically reduced by exploiting locality. Some studies show that there is sufficient temporal locality, i.e., an IP address may be repeatedly referenced over a period of time [1][2]. In earlier works full 32-bit IP address is used for caching [3], which exploits only temporal locality. On the other hand prefix caching exploits spatial locality by caching routing prefixes on the basis that a set of IP addresses in a network can be repeatedly referenced over a period of time. Such prefix cache is expected to show higher hit ratio than an IP address cache. Some works adopt a cache which consists of multiple zones, but optimal allocation of each zone and replacement policy remains unsolved.

Prefix caching has a drawback that a non-leaf prefix cannot be cached in itself, in spite of its high efficiency. If a prefix with one or more children prefixes is allowed to be cached, the cache hit result may or may not be the longest matching prefix. It does not seem that it is viable option to cache all of its children for ensuring longest matching. There are two alternative ways to cope with such problem. One is that no non-leaf prefix should be cached. The other is that non-leaf prefixes should be expanded to have no child prefix. The former approach restricts cache hit ratio while the latter increases the size of a routing table, cache miss penalty and update overhead.

Most previous researches on prefix caching focuses on non-leaf prefix expansion, which incurs the increase of routing table entries. When a cache miss occurs, the larger routing table takes more search time than the original one. Also, in case of insertion of a new prefix or deletion of a prefix, it is not easy to update the routing table with expanded prefixes. The design of our prefix cache aims at 1) exploitation of locality as much as possible 2) minimizing overhead to handle non-leaf prefixes 3) not only low miss ratio but low miss penalty, and 4) easy maintenance and updatability of a routing table.

The rest of this paper is organized as follows. Section 2 describes previous works related to caching schemes for IP lookup and also introduces prefix expansion schemes for caching. In section 3 we explain the proposed bitmap-based prefix caching scheme in detail. Section 4 describes how the bitmap in the routing table and the prefix cache can be managed in our scheme. In section 5 the performance of our scheme is evaluated using simulation, and we finally conclude the paper in section 6.

2. Background

2.1 Related Work

There have been many researches on exploiting the caching mechanism to speed up IP address lookup. Those caches can be classified into a few techniques: IP address caching, prefix caching, multi-zone caching, and short-cut caching.

The IP address cache stores 32-bit (or 128-bit for IPv6) IP addresses. Chiueh and Pradhan have presented a high speed IP lookup algorithm to utilize the cache in general-purpose CPU [3]. That scheme achieves IP caching by mapping IP addresses to virtual addresses. It uses two data structures, a destination host address cache (HAC), and a destination network address routing table (NART). It exploits only a temporal locality by caching destination IP address for future reuse. They expand their idea to exploit some degree of the spatial locality by caching host address range using shift address bits for indexing [4].

Liu has proposed prefix caching in which destination network address is cached, instead of individual destination IP address [5]. The prefix caching exploits spatial locality by caching a network address, i.e., a prefix which represents a range of IP addresses. It can drastically reduce cache miss ratio compared to IP caching. Unfortunately, it induces the incorrect result in case of a non-leaf prefix. To guarantee the correct lookup result, he presented three methods: complete prefix tree expansion (CPTE), no prefix expansion (NPE), and partial prefix tree expansion (PPTE). CPTE and PPTE increase the size of routing table, and it is hard to update an expanded table in these methods. NPE does not change the routing table, but cannot cache any non-leaf prefix.

Akhbarizadeh and Nourani have presented a new prefix caching scheme called reverse routing cache(RRC) to expand a prefix on the fly without any modifications to the original routing table [6]. They developed two approaches to deal with the non-leaf prefix: RRC-PR (RRC with Parent Restriction) and RRC-ME (RRC with Minimal Expansion). In RRC-PR, only disjoint prefixes except for a non-leaf prefix are cached. On the other hand, a non-leaf prefix is represented by the shortest expanded child and it is not overlapped with any other prefix in RRC-ME.

Zhu *et al*. have proposed an active routing prefix caching algorithm based on prefixes covering relationships [7]. A cache is partitioned according to the level of the prefix. It reduces the number of memory movements during the cache replacement. Although this algorithm excludes the enlargement of routing table, the descendent prefixes of the matching prefix should be cached together. Moreover, multiple prefixes may be matched in the cache. Meanwhile, Huang *et al*. have presented greedy prefix caching scheme to cache the non-leaf prefix as well as the expanded prefixes by RRC-ME to improve the performance [8]. It produces multiple matches in a cache. So, these schemes require the priority encoder in addition.

Zhang *et al*. have developed a scheme to cache the most used prefixes based on prediction [9]. They used an active cache memory and a standby cache memory to predict the popular prefixes. However, the prediction has a limit in accuracy. Many prefixes to be overlapped with popular prefixes should be cached in addition, in order to avoid incorrect result.

In multi-zone caching, a cache is divided into multiple sections mainly based on the matching prefix length. Locality may be less utilized due to traffic from a variety of sources in a single unified cache. On the other hand, multi-zone caching makes better use of locality in the way that the most likely used entries are located in larger section of cache. Shyu *et al*. have developed aligned-prefix caching(APC) based on singleton information to divide a cache into

two zones: a singleton-24 prefix cache and a singleton-32 prefix cache zones [10]. They also developed aligned-ancestor poisoning(AAP) to resolve non-cacheable prefix problem. Chvets and MacGregor split a cache into two zones where IP addresses are cached depending on the length of the matching prefix [11].

Kasnavi *et al*. have proposed a new scheme called short prefix expansion(SPE) to expand only short prefixes less than 17, which reduces overhead for the prefix expansion [12]. They also developed the multi-zone pipelined cache(MPC) to apply this expansion scheme to the concept of multi-zone caching. In MPC, the prefix caching is used for short prefixes, while IP caching is used for long prefixes.

Besides prefix caching there are several mechanisms to cache short-cut information to reduce the IP lookup latency. Peng *et al*. have devised a supernode caching scheme to decrease the number of memory accesses [13]. The original routing table should be constructed as a supernode tree based on the tree bitmap structure. This scheme caches recently visited supernode for the longest matching prefix. Ravinder *et al*. have proposed two-level cache structure to reduce IP lookup time [14]. It is composed of the prefix cache in the first level and the dynamic substride cache (DSC) in the second level. The DSC caches substrides, while the prefix cache stores the matching prefixes. On the prefix cache miss, the DSC is looked up for the substride corresponding to the IP address. The DSC can be used as the second level cache, regardless of the prefix cache structure.

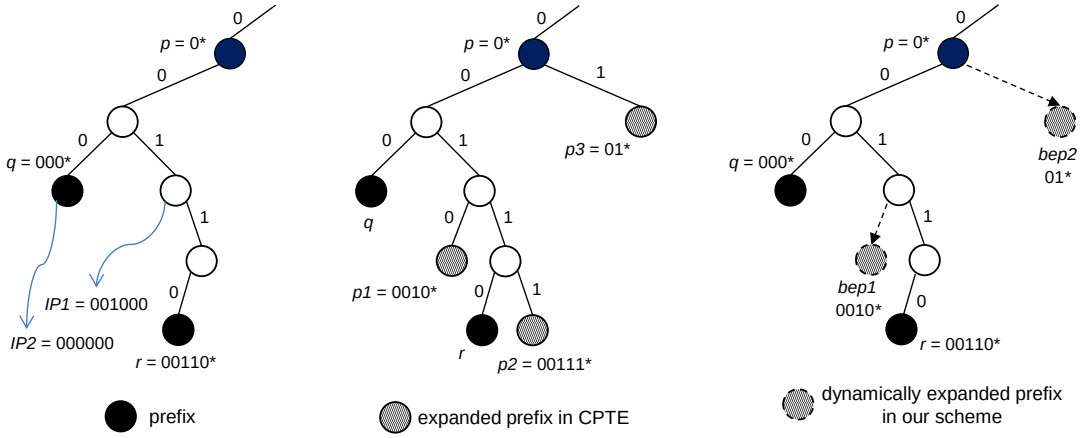
2.2 Prefix Expansion for Caching

It is obvious that a prefix cache shows lower miss ratio than an IP address cache with the same number of entries because a prefix can be substituted for a range of IP addresses. However, prefix caching has a drawback that not every prefix is cacheable. In case that a non-leaf prefix is cached, it is not guaranteed that the cache hit result is always correct.

Fig. 1(a) depicts an example of a trie in a routing table to show how such a wrong result occurs. For a given IP address *IP1*=001000, prefix *p*=0* is the matching result and *p* is cached for future reuse. The length of an IP address is assumed to be 6 bits instead of 32 bits for convenience. If next incoming IP address is *IP2*=000000, then *p* will be hit again in the cache. However, LMP in the routing table is *q*=000*, so *p* is not the correct result. Such *false hit* results from caching a non-leaf prefix. To prevent false hit in the cache, we have to allow only leaf prefixes to be cached, or the non-leaf prefix should be cached together with all of its descendants. It is not easy to manage such descendants in the cache, and even it requires prefixes to be ordered in the cache because it is made of TCAM.

While NPE simply does not allow non-leaf prefixes to be cached, CPTE changes from all the non-leaf prefixes to leaves by prefix expansion. Prefix expansion means converting a prefix into more longer prefixes which still cover the same address space [15][16]. **Fig. 1(b)** shows the routing table expanded by CPTE [5]. The non-leaf prefix *p* is expanded to three prefixes, *p*1, *p*2 and *p*3. Since the prefix *p* is finally removed after expansion, there remains no non-leaf prefix in the routing table. Note that all leaf prefixes including expanded prefixes completely cover a range of addresses represented by *p*. Unfortunately, CPTE increases the size of a routing table by around 50% due to expanded prefixes. CPTE is not viable due to the size of a routing table as well as difficult manageability due to expanded prefixes. Liu presented not only CPTE but also PPTE [5]. PPTE expands a non-leaf prefix down to the first level. For example, PPTE partially expands prefix *p* to *p*3 = 01*. In case of PPTE, the non-leaf prefix *p* remains in the routing table and it is marked as non-cacheable. In case that non-cacheable prefix is matched, 32-bit full IP address is cached instead. Since there are a number of non-cacheable prefixes, the cache miss ratio is relatively high.

Fig. 1(c) illustrates bitmap-based prefix expansion we propose in this paper. Bitmap-based prefix expansion does not increase a routing table unlike CPTE, and considerable number of prefixes are still effectively cached. When a non-leaf prefix is matched in the routing table, at most *k*-bit (3-bit in **Fig. 1(c)**) is dynamically expanded and the expanded prefix will be cached. In **Fig. 1(c)**, a non-leaf prefix *p* can be dynamically expanded to new prefix *bep1* = 0010* for an IP address 001000, and it is cached without causing false hit. For another IP address 011000, *p* is dynamically expanded to *bep2* = 01*. For expansion, every non-leaf prefix has a bitmap to represent the possibility of the expansion. We describe in detail how it is expanded in the following section.



(a) Original trie (b) CPTE (c) Bitmap-based expansion

Fig. 1. Prefix expansion schemes

3. Proposed Caching Scheme with Bitmap

3.1 Bitmap-based Prefix Expansion

No non-leaf prefix is cacheable because it incurs false hit. We expand a non-leaf prefix using its bitmap when it needs to be cached. Every prefix has a bitmap in our scheme. In this paper an expanded prefix means one of the prefixes which can be obtained by prefix expansion.

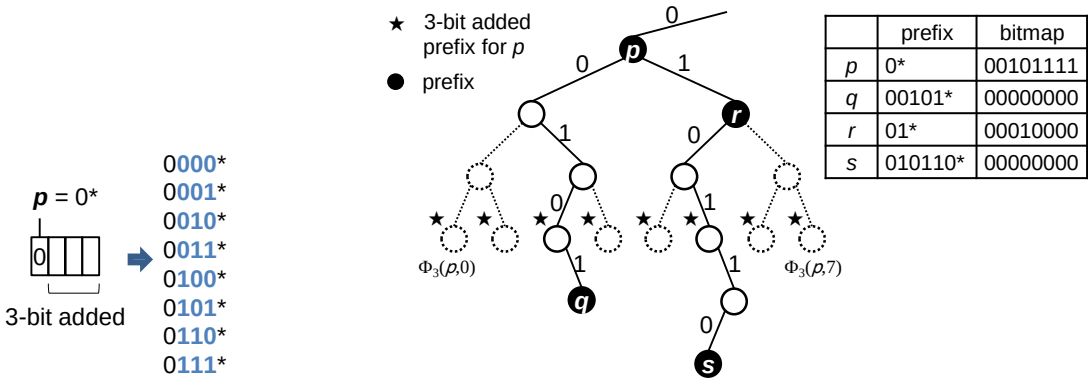
Definition 1. A *k-bit added prefix* for prefix *p* is one of the *k*-bit longer bit strings for *p* which are constructed by adding *k* bits to *p*. The *i*-th *k*-bit added prefix for prefix *p* is denoted by $\Phi_k(p, i)$, $0 \leq i \leq 2^k\text{-}1$.

The number of *k*-bit added prefixes for a prefix is 2^k. For example, all of the 2-bit added prefixes for 011* are 01100*, 01101*, 01110* and 01111*. Also, $\Phi_2(011*, 0)$ = 01100* and Φ_2 (011*, 3) = 01111*. It is not concerned whether there already exists any prefix between a prefix and its *k*-bit added prefix.

Definition 2. The *bitmap* of prefix *p* is a series of bits, $b_0b_1 \ldots b_{m-1}$ denoted by *bm*(*p*), where $m = 2^D$ and *D* is the *degree of expansion*. Each bit b_i represents whether the associated *k*-bit added

prefix for p has at least one lineal prefix longer than p. If $\Phi_D(p, i)$ has a descendant or an ascendant prefix (including itself) in a routing table, then $b_i = 1$. Otherwise, $b_i = 0$.

There are eight 3-bit added prefixes for prefix p=0* in **Fig. 2**, assuming the degree of expansion is 3. $\Phi_3(p, 2)$ and $\Phi_3(p, 5)$ have descendant prefixes which are q=00101* and s=010110* respectively. $\Phi_3(p, 4)$ to $\Phi_3(p, 7)$ have the ascendant prefix r=01* longer than p. So the associated bits b_2, b_4, b_5, b_6 and b_7 are all 1's, and $bm(p) = b_0b_1\ldots b_7 = 00101111$.



	prefix	bitmap
p	0*	00101111
q	00101*	00000000
r	01*	00010000
s	010110*	00000000

(a) 3-bit added prefixes (b) bitmaps of prefixes

Fig. 2. Bitmap-based prefix expansion

Definition 3. For a given IP address ip and its longest matching prefix p, the *bm-expanded prefix* is defined as follows. Suppose the k-bit added prefix matching with ip is q and its associated bit among $bm(p)$ is b_i. If b_i is 0, then the bm-expanded prefix is q. Otherwise it is ip itself.

In **Fig. 2,** p is the longest matching prefix for a given IP address ip_1=001111. Since the 3-bit added prefix matching with ip_1 is $\Phi_3(p, 3)$=0011* and its associated bit b_3=0, the bm-expanded prefix will be 0011*. For ip_2=001000, the 3-bit added prefix is $\Phi_3(p, 2)$=0010* and $b_2 = 1$, so the bm-expanded prefix will be ip_2 itself.

Postulate 1. A prefix or an expanded prefix is *cacheable* if it has no descendant prefix.

False hit occurs only when non-leaf prefix is cached. So any leaf prefix, i.e., which does not have any descendant, is safely cached.

Theorem 1. Every bm-expanded prefix is cacheable.
Proof. Suppose r is the bm-expanded prefix for a given IP address ip and its longest matching prefix p. r is either ip or a k-bit added prefix by definition 3. If r is ip, then it is full 32-bit and trivially cacheable. If r is the k-bit added prefix, then its associated bit b_i should be 0 by definition 3. It implies that k-bit added prefix does not have any descendant prefix by definition 2. So r is cacheable. □

It is worthwhile to remark that it is desirable to cache as short a prefix as possible. A shorter prefix covers a larger range of IP addresses and such prefixes occupy less cache entries. It is straightforward to get a shorter bm-expanded prefix by using a (k-1)-bit added prefix instead of a k-bit added prefix. If $\Phi_k(p, i)$ is a bm-expanded prefix and $b_i b_{i+1} = 00$ for even number of i, then we can also use $\Phi_{k-1}(p, i/2)$ as the bm-expanded prefix. That bm-expanded prefix is also cacheable. Similarly, $\Phi_k(p, i)$ is replaced by $\Phi_{k-1}(p, \lfloor i/2 \rfloor)$ as the bm-expanded prefix for odd number of i, when $b_{i-1} b_i = 00$. For example, in **Fig. 2**, the bm-expanded prefix is originally 0000* for a given IP address $ip_3 = 000010$, but 000* is also cacheable because the associated bits $b_0 b_1 = 00$. From (k-2)-bit added prefix to 1-bit added prefix can be applied in the same way without loss of generality.

If we use a higher degree of expansion, the efficiency of cache increases higher. For example, in **Fig. 2**, if we use 4 as the degree of expansion, 00100* can be cached for a given $ip_2 = 001000$, instead of ip_2 itself. In case the degree of expansion becomes 32, the bm-expanded prefix is actually the same as the matching prefix in CPTE. There is a trade-off between the efficiency of cache and the size of a bitmap when we increase the degree of expansion. For a given degree of expansion k, the size of a bitmap would be 2^k bits. We tentatively use 3 as the degree of expansion because the size of a bitmap is merely one byte.

3.2 Bitmap-based Prefix Cache

The overall architecture having bitmap-based prefix cache (BMCache) and a routing table is shown in **Fig. 3**. BMCache consists of memory and a simple prefix expansion logic. In BMCache bm-expanded prefixes and output ports are stored. Especially, the bm-expanded prefixes are stored in TCAM, so those can be searched in parallel. The corresponding output port number is stored in SRAM. In **Fig. 3** the routing table contains whole set of prefixes and it is looked up on cache miss. Each entry of the routing table contains a prefix and corresponding port number together with bitmap information. The bitmap-based prefix expansion can be performed only with a bitmap irrespective of lookup schemes.

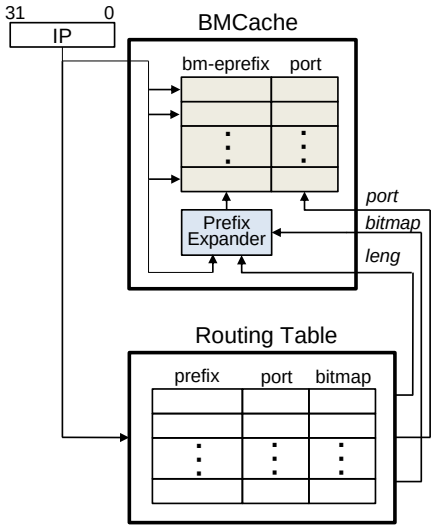


Fig. 3. Overall architecture

A prefix consists of a 32-bit IP address and its length. When IP address lookup is performed in the routing table, <*leng*, *port*, *bitmap*> is returned as the result where *leng* is the length of the matching prefix. IP address is not extracted from the routing table because BMCache is already provided with it. In BMCache, the prefix expander constructs a bm-expanded prefix using *leng*, *bitmap* and IP address.

Lookup

An overall lookup procedure is described in Fig. 4. For a given 32-bit IP address, TCAM of BMCache is searched in parallel (step 1). Note that there is only one match, if any, because any bm-expanded prefix has no descendant (see Theorem 1). The output port of the matched entry will be the lookup result (steps 1 and 2).

On cache miss lookup will be performed in the main routing table, and output port, bitmap and the length of the matched prefix is returned as a result (step 5). To construct a bm-expanded prefix the bitmap and the length of the matching prefix are used along with the given IP address by *Prefix Expander*. Then, the constructed bm-expanded prefix is stored into the TCAM of BMCache.

Procedure 1. LOOKUP(*IP*)

```
1: idx ← BMCache.lookup(IP)
2: if hit then    // Cache hit
3:         return BMCache.port[idx]
4: else           // Cache miss
5:         <leng, port, bitmap> ← RoutingTable.lookup(IP)
6:         replace(IP, <leng, port, bitmap>)
```

Fig. 4. Procedure of IP lookup

Cache Replacement

Fig. 5 shows how bm-expanded prefix can be constructed. The input, *leng*, is the length of the matching prefix and we finally obtain the length of bm-expanded prefix from that. The length of the bm-expanded prefix is 3-bit longer than the original matching prefix, or just 32-bit long. It can be resolved using bitmap. For a given IP, $i_0 i_1 \ldots i_{31}$, Selector extracts 3 bits, $i_{leng-1} i_{leng} i_{leng+1}$ which is used to choose a specific bit of *bitmap*. The length of the bm-expanded prefix is finally used to construct mask bits which will be stored into TCAM along with 32-bit IP address.

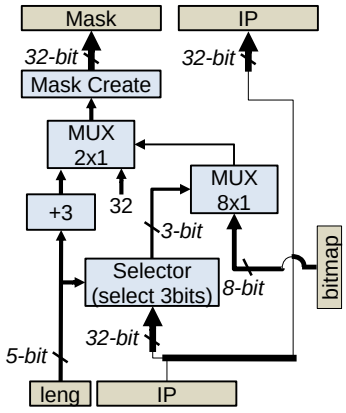


Fig. 5. Prefix expander

4. Bitmap Management in Routing Table

4.1 Initial Construction of Bitmap

Our bitmap-based scheme does not need any additional prefix entry such as the expanded prefix in a routing table. It requires just bitmap information field in each prefix entry in addition, as shown in **Fig. 3**. This subsection describes the method to make the bitmap information in the initial construction stage of the routing table. In this paper we explain the construction method for the routing table which is based on the binary trie, but that method can be similarly applied to other routing table structures.

The bitmap information of a prefix can be constructed by recursively checking the existence of children nodes up to the degree of expansion. **Fig. 6** shows an algorithm to construct a bitmap for given *node* and *height*. To construct the 2^D–bit bitmap for a given prefix *p*, Make_BM(*p*, *D*) should be called where *D* is the degree of expansion.

Procedure 2. Make_BM(*node*, *height*)
// *D* is the degree of expansion
// "a || b" denotes the concatenation of a and b.
1: **if** *node* is null **then return** 2^{height}-bit 0's
2: **if** *node* is a prefix and height < *D* **then return** 2^{height}-bit 1's
3: **if** *height* is 0 **then return** 1
4: **return** Make_BM(*node*->left_child, *height*-1) || Make_BM(*node*->right_child, *height*-1)

Fig. 6. Procedure of bitmap construction

Fig. 7 illustrates an example for Proc. 2. The 8-bit bitmap of prefix *p* can be constructed by means of Make_BM(*p*, 3), which is computed by concatenation of Make_BM(*a*, 2) and Make_BM(*r*, 2). Make_BM(*a*, 2) and Make_BM(*r*, 2) are also computed recursively. The recursion will stop when it reaches a prefix or null, or the height comes to 0. Since *r* is a prefix, Make_BM(*r*, 2) just returns all 1's, i.e., 1111 without further recursion as described in step 2 of Proc. 2. In case it reaches null, a 2^{height}-bit 0's is returned. For example, Make_BM(*a*->left_child, 1) is 00 and Make_BM(*b*->right_child, 0) is 0. If the height of a node becomes 0 and it is not a prefix node, the node must have at least one descendant prefix. So, in case that the height reaches 0, bit 1 should be returned as step 3 of Proc. 2. For example, Make_BM(*c*, 0) returns 1 in **Fig. 7**. Note that the node *c* has a descendant prefix *q*.

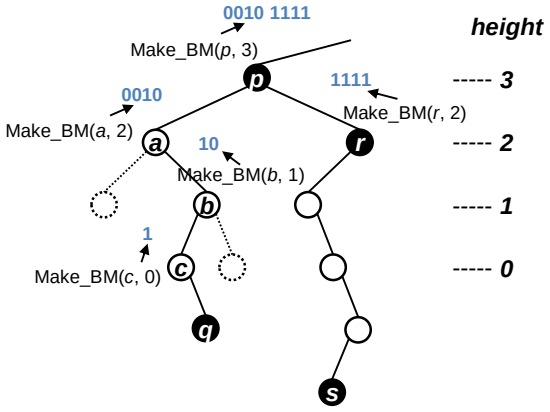


Fig. 7. An example of bitmap construction

The bitmap of each prefix p in a routing table is constructed by calling Make_BM(p, D). It produces several recursive calls, which cause at most (2^{D+1}-2) accesses to nodes. The recursion will stop at null or a prefix, so the number of memory accesses is actually total number of nodes up to immediate descendant prefixes. If there are n prefixes in a routing table, the total time complexity for bitmap construction is $O(2^{D+1} \cdot n)$ in worst case. The degree of expansion, D, is a very small constant and typically less than 4, so 2^{D+1} is thought of as a negligible constant value.

4.2 Updating Bitmap in Routing Table

The routing table is frequently updated due to prefix insertion and deletion. In this subsection we explain how bitmaps are updated when a prefix is inserted or deleted. It is clear that the prefix deletion does not need bitmap construction regarding that prefix while the prefix insertion requires the construction of its bitmap. Since the bitmap of a prefix reflects the existence of its descendants, the insertion or the deletion of a prefix p may give rise to the change of the bitmap of p's parent prefix. However, the descendants of p are not affected by insertion and deletion of p. The ascendants of p except p's parent do not need to change their bitmaps either.

The update of a routing table itself depends on the underlying structure, and it is beyond the scope of this paper. Note that a lookup scheme can be separated from the update management. Even though the algorithms for bitmap update are described in trie-based table, the lookup scheme is not limited to the trie-based scheme.

In **Fig. 8**, Proc. 3 describes the algorithm to update bitmaps on the insertion of a new prefix *np*. The bitmap for *np* is newly generated by calling Make_BM(*np*, *D*) in step 1, and also the bitmap for its parent prefix *par* is updated in steps 2~5. After finding the parent prefix *par*, it sets the bits in the bitmap where each of the corresponding *D*-bit added prefixes is a lineal prefix of *np*. Note that it does not require additional overhead to find the parent prefix, because ancestors are already determined while it locates the prefix *np* for inserting.

Procedure 3. Update_Bitmap_Insertion(*np*)
// *np* is a newly inserted prefix.
// *D* is the degree of expansion.
1: generate the bitmap of *np* by Make_BM(*np*, *D*)
2: let *par* be the parent prefix of *np*
3: let $b_0\ b_1 \ldots b_L = bm(par)$ where L is 2^D-1
4: set b_i for each i such that
5: $\Phi_D(par,i)$ is a descendant or an ascendant of *np*, or *np* itself

Fig. 8. Bitmap update procedure for the prefix insertion

It is straightforward to find the k-bit added prefixes which are the descendants or the ancestor of *np*. After such k-bit added prefixes are determined, all the corresponding bits will be set in the bitmap. Suppose that a prefix x=000* is inserted in **Fig. 9**. 0000* and 0001* are the descendants of x, which are also 3-bit added prefixes for parent prefix 0*, i.e., $\Phi_3(p, 0)$ and $\Phi_3(p, 1)$ respectively. So the bits b_0b_1 in the bitmap of prefix p should be set as 1, and the bitmap is changed from 00111011 to **11**111011.

As described above, there is no additional overhead to find the parent of the inserted prefix. So the number of memory accesses for bitmap update on prefix insertion is only affected by

Make_BM(*np*, *D*) in step 1 of Proc. 3. The total number of memory accesses is at most 2^{D+1}-2, however, inserting a leaf prefix does not incur any extra memory accesses for bitmap update. In worst case the number of memory accesses is 14 for *D*=3.

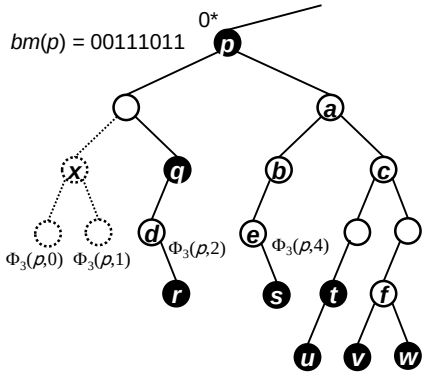


Fig. 9. The examples for bitmap updates

In order to explain the bitmap update algorithm on the prefix deletion, we introduce a new terminology, a *bridge node*.

Definition 4. The *bridge node* of a leaf prefix *p* is the closest ancestor node of *p* which can remain in the trie after the deletion of *p*.

In **Fig. 9,** the bridge node of *r* is *q*, because node *d* disappears but *q* still remains after the deletion of *r*. Similarly, nodes *b* and *e* disappear on deletion of *s*, so the bridge node of *s* is *a*.

An algorithm to update bitmap on deletion of a prefix is described in Proc. 4 of **Fig. 10**. At first, it tries to find the parent prefix *par* of the deleted prefix *op*, and then the distance between *op* and *par* is compared with the degree of expansion *D*.

Procedure 4. Update_Bitmap_Deletion(*op*)

```
//  op is a deleted prefix.
//  |p| is the length of prefix p.
1: let par be the parent prefix of op
2: if |op| - |par| ≤ D then
3:     str ← Make_BM(op, D - (|op| - |par|))
4:     update b_x…b_y in bitmap bm(par) as str, where
5:           Φ_D(par,i) is a descendant of op, x ≤ i ≤ y
6: else
7:     if op is a leaf prefix then
8:         let brg be the bridge node of op
9:         if |brg| - |par| < D then
10:            reset b_i in bm(par), where
11:              Φ_D(par,i) is an ancestor of op
```

Fig. 10. Bitmap update procedure for the prefix deletion

If the distance (|*op*|-|*par*|) is not greater than *D* in step 2 of Proc. 4, it updates *bm*(*par*), especially the bits corresponding to descendants of *op*. The new bit string *str* can be obtained

by calling Make_BM(*op*, *D*-(|*op*|-|*par*|)). Suppose *q*=001* is deleted in **Fig. 9**. Since |*q*| - |*p*| = 2 < 3 (=*D*), Make_BM(*q*, 1) is called and it returns 10. $\Phi_3(p, 2)$ and $\Phi_3(p, 3)$ are descendants of *q*, so b_2b_3 is replaced with new string 10. Now the bitmap of *p* is changed from 00111011 to 00**10**1011.

In case that the distance (|*op*|-|*par*|) is greater than *D* in step 6 of Proc. 4, *bm*(*par*) is updated only when *op* is a leaf prefix. If *op* is a non-leaf prefix, deletion of *op* does not affect *bm*(*par*). For example, the deletion of a non-leaf prefix *t* does not affect the bitmap of *p* in **Fig. 9**. In case that the deleted prefix *op* is a leaf prefix, the distance between *brg* and *par* is computed where *brg* is the bridge node of *op*. If the distance (|*brg*|-|*par*|) is less than *D* in step 9 of Proc. 4, it determines $\Phi_D(par, i)$ which is the ascendant of *op*. Since $\Phi_D(par, i)$ will be removed after the deletion of *op*, the associated bit b_i in the bitmap should be changed from 1 to 0. Suppose that a prefix *s*=01001* is deleted in **Fig. 9**. The bridge node of *s* is *a* and the parent prefix of *s* is *p*. Therefore, |*a*| - |*p*| = 1 < 3 (=*D*). b_4 in *bm*(*p*) should be set to 0 because $\Phi_3(p, 4)$ is the ascendant of *s*. So the updated bitmap becomes 0011**0**011. Now suppose a prefix *v*=011100* is deleted in **Fig. 9**. The bridge node of prefix *v* is *f* and the parent prefix of *v* is *p*. Since |*f*| - |*p*| = 4 > 3 (=*D*), the bitmap of prefix *p* is not affected.

The bridge of the deleted prefix can be determined without additional overhead in the same way as the parent prefix. So the number of memory accesses for bitmap update on prefix deletion is only affected by Make_BM(*op*, *D*-(|*op*|-|*par*|)) in step 3 of Proc. 4. Since |*op*|-|*par*| is at least 1, the total number of memory accesses is at most 2^D-2. It implies that the bitmap update cost for prefix deletion is merely 6 in worst case for *D*=3.

4.3 Cache Coherency for Update

The insertion or deletion of a prefix may give rise to invalidate expanded prefixes in BMCache. Suppose that an expanded prefix 0101* has been cached in **Fig. 9**. If a prefix 010* (or 01011*) is inserted to the routing table, the prefix 0101* in BMCache may cause incorrect results. For correct IP lookup, that prefix should be removed from BMCache.

On insertion of a new prefix *np*, the bm-expanded prefixes for the parent prefix *par* of *np* should be considered for cache coherency. It should be checked which bit in the bitmap of *par* is changed from 0 to 1. For each changed bit, the corresponding bm-expanded prefix should be invalidated in BMCache.

BMCache may contain an IP address itself as a bm-expanded prefix as described in section 3.1. In **Fig. 9**, for a given IP address 010001, the matching prefix is *p*=0* but cannot be expanded to $\Phi_3(p, 4)$ because b_4 in the bitmap of *p* is 1. In that case the IP address itself is cached in BMCache. Now, suppose that prefix 010* is inserted. Then, the cached IP address must be removed from BMCache. Actually, all the IP addresses which are descendants of the inserted prefix may be removed for simplicity. The IP addresses which are not matched with the inserted prefix can be removed from BMCache as well, but it does not cause incorrect results.

On the other hand, the deletion of prefix *op* from a routing table may cause removal of prefix *op* itself or its expanded prefixes in BMCache. Also, all the IP addresses covered by *op* may be removed from BMCache in the same way as the insertion. For a given IP address 010001, in **Fig. 9**, the IP address itself is cached in BMCache. On deletion of prefix 0*, all the descendant IP addresses including 010001 may be removed from BMCache.

5. Performance Evaluation

For our experiment, the routing tables are constructed for each cache scheme using a real routing table in [17]. **Table 1** shows the number of prefixes in each cache scheme. For BMCache, there is no additional entry compared to original table. We use 8-bit bitmap, so there are possibly up to eight 1's in bitmap. **Table 2** shows non-leaf prefix count according to the number of 1's in bitmap. The bitmap having fewer 1's may result in better performance. In almost half of total non-leaf prefixes the number of bits valued at 1 is less than 5.

Table 1. The number of prefixes for cache scheme

	BMCache	CPTE	PPTE
Leaf	334,166	334,166	334,166
Non-leaf	34,438	34,438	34,438
Expanded	0	169,930	13,433
Total	368,604	538,534	382,037

Table 2. Distribution of non-leaf prefix count according to the number of 1's in bitmap

	1	2	3	4	5	6	7	8	Total
Count (ratio)	5,409 (0.16)	4,730 (0.14)	1,771 (0.05)	5,169 (0.15)	1,341 (0.04)	1,847 (0.05)	1,112 (0.03)	13,059 (0.38)	34,438 (1.00)

All publicly available traces are anonymized over IP addresses for privacy, so the generated traces are used for our experiment instead of real traces. Anonymized traces maintain spatial and temporal locality of the original traces, but those include a lot of non-matching packets for real prefixes. We generate traces in which every IP address matches with some prefix in a real routing table, while they completely reflect the spatial and temporal locality of real traces. The characteristics of the localities are extracted from four real traces in CAIDA [18]. Note that the cache performance is highly dependent on those localities.

Figures 11-14 compare cache miss ratios of six schemes using the generated traces (named Trace-1, Trace-2, Trace-3 and Trace-4). In those figures 'IPcache' represents a 32-bit full IP address cache. Since 'IPcache' stores fixed-length IP addresses, it is sufficient to use CAM instead of TCAM. The number of transistors in a TCAM cell is about two times more than that in a CAM cell. So we assume it has twice as many entries as other schemes for the same size of memory. Similarly, it is assumed that MPC has twice entries, because MPC consists of two parts, short prefix cache and IP cache, each of which occupies half size of the prefix cache for the same number of entries. In the experiments 8-bit bitmap is used for BMCache, i.e., the degree of expansion is 3. The simulation results show that the miss ratio of CPTE is generally lower than those of any other schemes. BMCache shows next lower miss ratio. Note that the routing table of BMCache has the same number of entries as original routing table because there is no expansion in the routing table unlike CPTE.

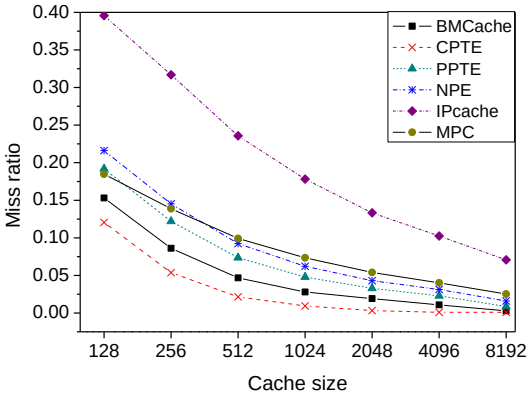


Fig. 11. Cache miss ratio (Trace-1)

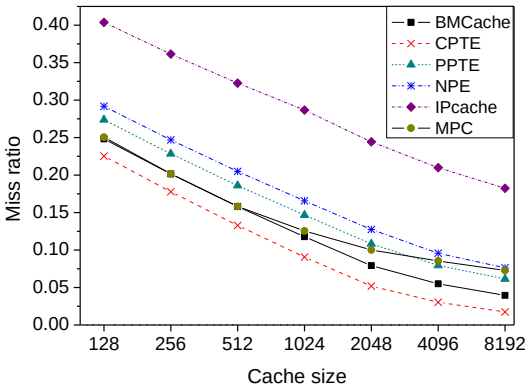


Fig. 12. Cache miss ratio (Trace-2)

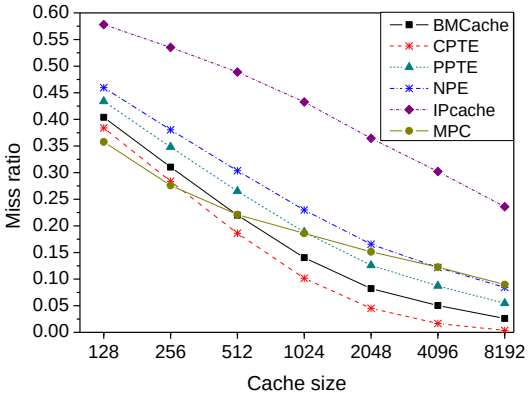


Fig. 13. Cache miss ratio (Trace-3)

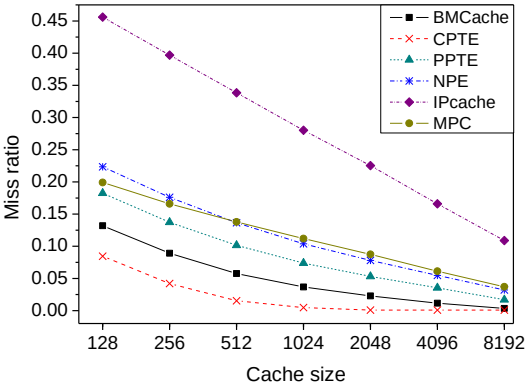


Fig. 14. Cache miss ratio (Trace-4)

Fig. 15 shows the miss ratio of BMCache according to the degree of expansion *D*. As *D* increases, the miss ratio becomes lower. However, the size of the bitmap will be doubled in the routing table whenever *D* increases. For example, the bitmap size is 16 bits for $D = 4$ while that is 8 bits for $D = 3$.

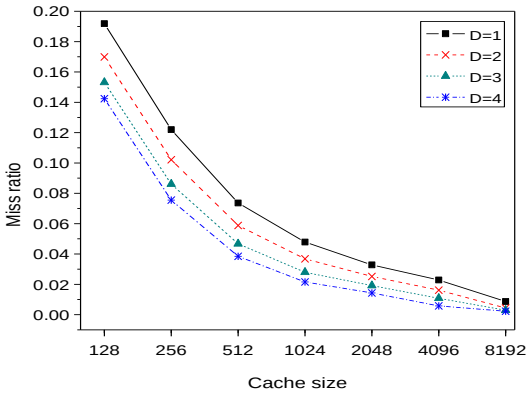


Fig. 15. Miss ratio according to the degree of expansion *D*

Though CPTE has lower miss ratio than BMCache, CPTE may have a great update overhead because complete expansion produces a large number of children. **Fig. 16** shows the number of expanded children for a non-leaf prefix. It shows some non-leaf prefixes have a

great number of expanded children. About 20% of non-leaf prefixes have more than 8 expanded children and even 0.4% of non-leaf prefixes have more than hundred expanded children.

Fig. 17 compares the update overhead of a non-leaf prefix in CPTE and BMCache. In CPTE insertion or deletion of a non-leaf prefix affects its immediate descendants and expanded children, so the overhead is estimated by total length of paths from a non-leaf node to its immediate descendants and expanded children. Meanwhile, in BMCache, there is no actual expansion in the routing table, so the update overhead of a non-leaf prefix is estimated for its bitmap by summing up the path length to immediate descendants, however, up to the degree of expansion D=3. In **Fig. 17** count of non-leaf prefixes is cumulated from the highest to the lowest update overhead. It shows the update overhead of CPTE is higher than BMCache.

Table 3 shows the number of the expanded children and update overhead in CPTE and BMCache. There is a great gap between CPTE and BMCache in the worst case update overhead. The update overhead of CPTE is no less than 8,680, while that of BMCache is merely 14. The update overhead incurs delay of lookup, so the forwarding engine should have sufficient resources to meet the worst case update overhead.

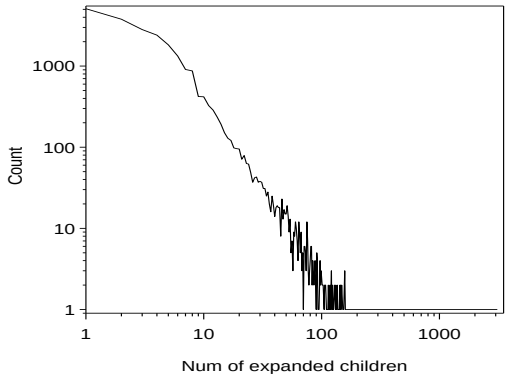


Fig. 16. Distribution of the number of the expanded children (CPTE)

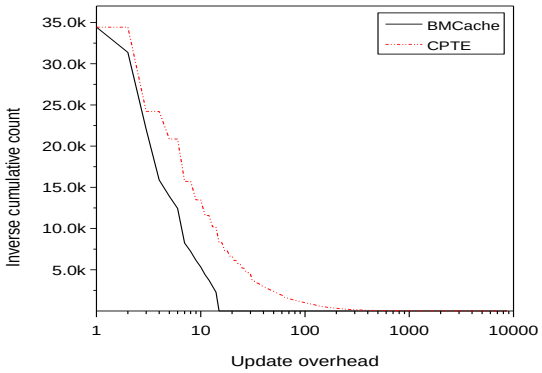


Fig. 17. Distribution of the update overhead

Table 3. Expanded children and update overhead

	Max.	Avg.
Expanded children (CPTE)	3,097	4.93
Update overhead (CPTE)	8,680	18.66
Update overhead (BMCache)	14	4.95

6. Conclusion

This paper proposes a new prefix caching scheme using bitmap, which shows high performance IP lookup as well as low update overhead. In that scheme every prefix has a bitmap in which each bit indicates whether the prefix can be expanded to some descendant or not. When an IP address is matched with a non-cacheable prefix in a routing table, its expanded prefix or the IP address itself is cached according to the corresponding bit in the bitmap. Using the bitmap any prefix can be effectively cached as the expanded one, which makes it possible to exploit a high degree of locality with low overhead. Since the proposed scheme does not increase the number of prefix entries in a routing table, cache miss penalty is

relatively low compared to other schemes because the lookup time usually increases as the number of prefixes increases in the routing table. Also, our scheme can update the routing table faster than any other prefix caching schemes owing to no additional prefixes and less update overhead of bitmaps in the routing table.

The performance of the proposed scheme is evaluated by simulation with the real forwarding table and a variety of traces reflecting the characteristics of locality in real traffics. The simulation result shows that the cache miss ratio is slightly higher than CPTE, but is better than other schemes such as PPTE, NPE, and MPC. In addition, it shows that the prefix update in our scheme requires much lower overhead than that in CPTE.

References

[1] W.-L. Shyu, C.-S. Wu, and T.-C. Hou, "Efficiency analyses on routing cache replacement algorithms," in *Proc. of IEEE International Conference on Communications (ICC)*, vol. 4, pp. 2232–2236, April–May 2002. Article (CrossRef Link).

[2] W. Shi, M. MacGregor, P. Gburzynski, "On temporal locality in IP address sequences," *IEICE Transactions on Communications*, E86-B (11), pp. 3352–3354, 2003. Article (CrossRef Link).

[3] T. Chiueh and P. Pradhan, "High performance IP routing table lookup using CPU caching," in *Proc. of IEEE conference on Computer Communication (INFOCOM)*, pp. 1421-1428, Mar. 1999. Article (CrossRef Link).

[4] T. Chiueh and P. Pradhan, "Cache memory design for network processors," in *Proc. of International Symp. on High Performance Computer Architecture*, pp. 409-419, Jan. 2000. Article (CrossRef Link).

[5] H. Liu, "Routing prefix caching in network processor design," in *Proc. of International Conf. on Computer Communications and Networks*, Oct. 2001. Article (CrossRef Link).

[6] M. J. Akhbarizadeh and M. Nourani, "Efficient prefix cache for network processors," *12th Annual IEEE Symp. on High Performance Interconnects*, pp. 41-46, Aug. 2004. Article (CrossRef Link).

[7] G.-s. Zhu, S.-h. Yu, and J.-y. Dai , "An Active Routing Prefix Caching Algorithm for IP Address Lookup," in *Proc. of International Conf. on ChinaCOM 2009*, pp. 1-6, Aug. 2009. Article (CrossRef Link).

[8] Z. Huang, G. Liu, and J.-K. Peir, "Greedy Prefix Cache for IP Routing Lookups," in *Proc. of 10th International Symposium on Pervasive Systems, Algorithms, and Networks (ISPAN)*, pp. 92-97, Dec. 2009. Article (CrossRef Link).

[9] W. Zhang, J. Bi, J. Wu, and B. Zhang, "Caching Popular BGP Prefixes with Grey Modeling Prediction," in *Proc. of 20th International Conference on Computer Communications and Networks (ICCCN)*, pp. 1-6, Aug. 2011. Article (CrossRef Link).

[10] W.-L. Shyu, C.-S. Wu, and T.-C. Hou, "Aligned prefix caching based on singleton information," *Computer Networks*, vol. 47, no. 6, pp. 871–884, 2005. Article (CrossRef Link).

[11] I. L. Chvets and M. MacGregor, "Multi-zone caches for accelerating IP routing table lookups," *Merging Optical and IP Technologies Workshop on High Performance Switching and Routng*, pp. 121-126, May 2002. Article (CrossRef Link).

[12] S. Kasnavi, P. Berube, V. Gaudet and J. N. Amaral, "A cache-based internet protocol address lookup architecture," *Computer Networks*, pp. 303-326, vol. 52, issue 2, Feb. 2008. Article (CrossRef Link).

[13] L. Peng, W. Lu, and L. Duan, "Power Efficient IP Lookup with Supernode Caching," in *Proc. of IEEE GLOBECOM '07*, pp. 215-219, Nov. 2007. Article (CrossRef Link).

[14] S. Ravinder, M.A. Nascimento, and M.H. MacGregor, "Two-level cache architecture to reduce memory accesses for IP lookups," in *Proc. of International Conference on Teletraffic Congress (ITC)*, pp. 278-285, Sep. 2011. Article (CrossRef Link).

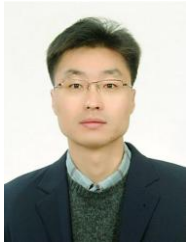
[15] V. Srinivasan and G. Varghese, "Fast Address Lookups Using Controlled Prefix Expansion," *ACM Trans. Computer Systems*, 1999. Article (CrossRef Link).

[16] M. A. Ruiz-Sanchez, Ernst W. Biersack, and Walid Dabbous, "Survey and taxonomy of IP address

lookup algorithms", *IEEE Network*, vol. 15, issue 2, pp. 8-23, March-April 2001. Article (CrossRef Link).
[17] APNIC prefix table, http://thyme.apnic.net/ap-data/2011/08/25.
[18] The CAIDA UCSD Anonymized Internet Traces 2011 – Aug 25 and May 19 http://www.caida.org/data/passive/passive_2011_dataset.xml.

Jinsoo Kim received the B.S. degree from Seoul National University, Seoul, in 1983, and the M.S. and Ph.D degrees from Korea Advanced Institute of Science and Technology (KAIST), in 1985 and 1998, respectively, all in computer engineering. In 1985, he joined Korea Telecom, where he was a senior researcher. In 2000, he joined the faculty of Konkuk University, where he is now a professor. His research interests include parallel computing architectures, high-speed networking, wireless sensor networks, and packet processing systems.

Myeong-Cheol Ko is a professor of computer engineering at Konkuk University, where he directs the AVIT(Advanced Visualization and Interaction Technology) research group. He received PhD in computer science from Yonsei University in 2003. His research interests are in 3D computer graphics and human-computer interaction focusing on the design and implementation of augmented reality systems.



Junghyun Nam received the B.E. degree in Information Engineering from Sungkyunkwan University, Korea, in 1997. He received his M.S. degree in Computer Science from University of Louisiana, Lafayette, in 2002, and the Ph.D. degree in Computer Engineering from Sungkyunkwan University, Korea, in 2006. He is now an associate professor in Konkuk University, Korea. His research interests include cryptography and computer security.

Junghwan Kim received the B.S., M.S. and Ph.D degrees from Seoul National University, Seoul, in 1991, 1993 and 1999, respectively, all in computer science. In 1999 he joined Samsung Electronics, where he was a senior researcher. In 2001 he joined the faculty of Konkuk University, where he is now a professor. His research interests are in the areas of parallel computing, communication networking, GPU computing, and design of efficient algorithms.