SMART

Smart Multi-Array Routing Table

Yoichi Hariguchi[1]

Allegro Systems, Inc.
yoichi@allegrosys.com

Abstract

It is known that the routing table based on multibit trie, in other words, multiple levels of arrays (hereafter let us call it MART, Multi-Array Routing Table) has a very low and deterministic search cost of modestly higher memory consumption compared to other routing table approaches. The search cost of MART is typically 2 to 4 routing table memory accesses for IPv4. MART has the additional benefits that it is easy to implement in hardware and its determinism allows for pipelining route lookups in hardware. The primary drawback of MART is that update operations, namely adding and deleting routes, are highly costly relative to the route lookup operation. In particular, MART has a problem at deletion which no paper has addressed. This paper proposes an extension to MART, called "Smart Multi-Array Routing Table" or just SMART, which gives a solution for this issue. SMART not only has the low cost and determinism of route lookups but also provides low cost route update operations, which always have lower than 256 routing table memory accesses regardless of both the number of routes in the routing table and the prefix length for IPv4.

[1]This work was performed while the author was with MAYAN Networks, Corp., San Jose, CA

1 Introduction

The size of the Internet routing table is growing rapidly [1] even after the introduction of CIDR (Classless Inter-Domain Routing) [2]. The number of routes in a core router is almost 100,000 [3] as of this writing. In addition, the routing instability [4] is becoming a serious problem. This problem is also called 'route flap'. The route flap often causes deleting and adding the entire set of BGP (Border Gateway Protocol) routes. It is important to enhance not only route lookup performance but also route update performance because slow route update may cause a route flap storm.

It is known that MART has a very low and deterministic search cost. For example, the route lookup cost of a MART implementation by Pankaj Gupta, Steven Lin, and Nick McKeown [5] is 2 memory accesses in the worst case. However, their MART implementation has a serious drawback, which is its highly expensive update cost. It needs 32M routing table memory accesses (16M reads and 16M writes) to add a single route in the worst case. Another MART implementation by Srinivasan and Varghese [6] has much better worst case update cost, but it still does not solve a problem of MART at route deletion. Actually, neither paper discusses the problem of route deletion at all. In this paper, the author first discusses the problem of MART at route deletion and then introduces a new routing table design called SMART (Smart Multi-Array Routing Table) that solves the problem. SMART is an extension of MART and its route update cost is always smaller than 256 routing table memory accesses (128 writes and 127 reads).

2 Previous Work

Figure 1 shows an example of simple MART. The MART in Figure 1 consists of 3 levels of arrays.

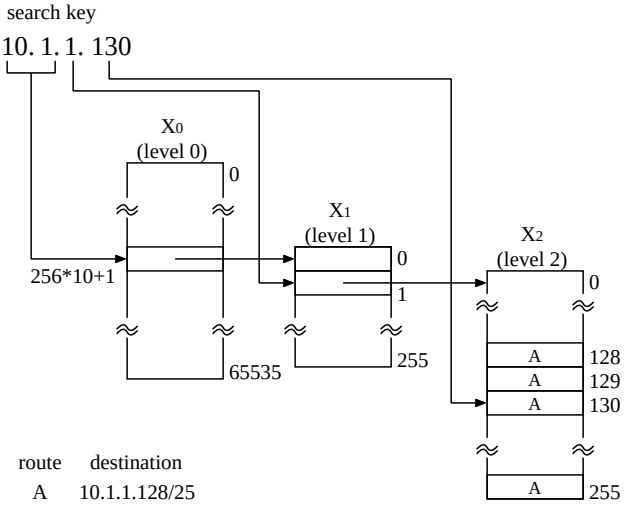


Figure 1: Simple Multi-Array Routing Table

The level 0 array X_0 is indexed by the most significant 16 bits of the search key IPv4 address and has 64K elements. The level 1 array X_1 is indexed by bit 8..15 of the search key IP address and has 256 elements. The level 2 array X_2 is indexed by the least significant 8 bits of the search key IPv4 address and has 256 elements. In other words, each IP address can be mapped to one element of the arrays of level 0, level 1, or level 2. Hereafter, let us call the bit length used to index an array

at each level *stride length* . and let S_i be the *stride length* at level i. In Figure 1, S_0 is 16, S_1 is 8, and S_2 is 8 respectively. Let W be the bit length of IP address ($W = 32$ in the case of IPv4) then $W = \sum_{i=0}^{maxLevel} S_i$.

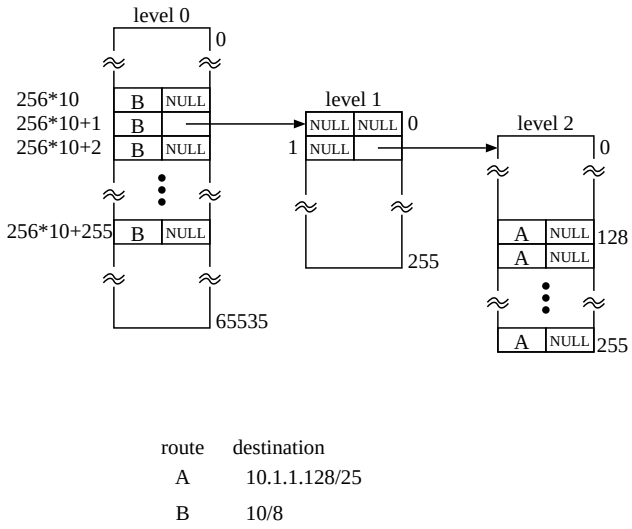
When a route is inserted, all of the array elements of the routing table that correspond to the destination IP prefix of the inserted route are set to point to that route. For example, the destination IP prefix of route A in Figure 1 is 10.1.1.128/25. That is why the elements from 128 to 255 in array X_2, which are corresponding to 10.1.1.128/25, have pointers to route A. The MART in Figure 1 always finishes a lookup within 3 routing table memory accesses.

Assume a new route 'route B' whose destination IP prefix is 10/8 is to be inserted to the MART in Figure 1. Figure 2 shows a pseudo code to add route B to the routing table in Figure 1. Figure 2 indicates that it takes 16M (256×256×256) routing table memory reads and 16M routing table memory writes to add one entry in the worst case.

```
(1)   for i = 10 × 256 + 1 to 10 × 256 + 255
(2)     if X0[i] is connected to a level 1 array then
(3)       Level 1 array X1 = X0[i]
(4)       for j = 0 to 255
(5)         if X1[j] is connected to a level 2 array then
(6)           Level 2 array X2 = X1[j]
(7)           for k = 0 to 255
(8)             if X2[k] == NULL or prefix length of the route pointed to
                by X2[k] < 8 then
(9)               X2[k] = B
(10)            endif
(11)          endfor
(12)        else if X1[j] == NULL or prefix length of the route pointed to
            by X1[j] < 8 then
(13)          X1[j] = B
(14)        endif
(15)      endfor
(16)    else if X0[i] == NULL or prefix length of the route pointed to by
        X0[i] < 8 then
(17)      X[i] = B
(18)    endif
(19)  endfor
```

Figure 2: Inserting Route 10/8 to MART in Figure 1

An idea called 'Controlled Prefix Expansion' in a paper by Srinivasan and Varghese [6] reduces the route update cost of MART. Hereafter, let us call their MART implementation MART-CPE. MART-CPE has two pointers per element. The one pointer (say $pCPE$) points to the longest-matching route associated with the element. The other pointer (say $pNext$) points to the next level array if it exists. The MART in Figure 1 and 2 spreads a pointer to the longest-matching route all over the routing table. In contrast, MART-CPE spreads a pointer to the longest-matching route only within an array. Figure 3 shows the MART-CPE routing table that has route A and route B. The reason MART-CPE does not have to spread a pointer all over the routing table is that MART-



route	destination
A	10.1.1.128/25
B	10/8

Figure 3: MART with Controlled Prefix Expansion

CPE prepares a variable called BMP and updates BMP with the value of $pCPE$ each time MART-CPE visits a deeper level array at route lookup. The route insertion cost of MART-CPE is 256 memory accesses in the worst case.

Assume now route A is to be removed. In the case of the routing table in Figure 1, the pointers of element 128 to 255 of the level 2 array must be replaced with the new longest-matching route after route A is removed, which is route B. In the case of the routing table in Figure 3, $pCPE$ of element 128 to 255 of the level 2 array must be replaced with the new longest-matching route after route A is removed, which is NULL. Neither the paper by Gupta, Lin, and McKeown [5] nor the paper by Srinivasan and Varghese [6] discusses how to find the new longest-matching route that replaces the route to be deleted. The simplest way is to backtrack through the array elements (and arrays in the case of MART). This process is expensive. The simple backtrack takes 254 memory accesses in the worst case even in MART-CPE.

SMART solves this backtracking problem of MART at route deletion. SMART always finds the route to be replaced with the route to be deleted at one memory access without any backtracking. In summary, the performance of three routing tables is as follows:

	Search	Insert	Deletion
MART	fast	slow	slow
MART-CPE	fast	fast	slow
SMART	fast	fast	fast

3 SMART

First, let us define some terminologies. The *base index* is the smallest array index that matches a route. For example, the *base index* of route A is 128. Say that index j is *above* index i if there are routes X and Y such that i is the *base index* of X, j is the *base index* of Y, and Y is a prefix of X, and $j \neq i$. The indices *above* i are obtained by removing 1 bits from right to left. For example,

the indices above 11 (00001011) are 10 (00001010), 8 (00001000), and 0. If index i has l 1 bits, there are $l - 1$ indices *above* i.

3.1 Data Structure

Figure 4 shows the array element structure of SMART.

Figure 4: SMART Array Element

A SMART array element i at level l has the following 3 pointers:

- $X_l[i].pLM$ points to the longest matching route whose *base index* is i, or NULL if there is no such route.
- $X_l[i].pDef$ points to the longest matching route whose *base index* is *above* i, or NULL if there is no such route. Let us call the route pointed to by $pDef$ element default route.
- $X_l[i].pNext$ points to the next level array corresponding to index i at level l.

Each route entry $rtEnt$ has at least the following fields:

- $rtEnt.dest$ indicates the destination IP address of the route
- $rtEnt.plen$ indicates the prefix length of the route
- $rtEnt.pNext$ points to the longest matching route whose *base index* is the same as that of $rtEnt$ but whose prefix length is shorter than $rtEnt{\rightarrow}plen$, or NULL if there is no such route.

The $rtEnt.pNext$ field is necessary to handle overlapping routes, which have the same destination IP address and the different prefix lengths. For example, 10/8, 10/9, 10/10, ... 10/32 are overlapping routes. Overlapping routes have the same *base index* since their destination IP addresses are the same. SMART makes a singly linked list of overlapping routes using $rtEnt.pNext$ field. The linked overlapping routes are sorted according to the descending order of their prefix lengths when the routes are inserted or deleted so that no extra search cost is introduced. Neither paper [5] nor paper [6] discusses how to process overlapping routes.

3.2 Route Insertion

Suppose we use a routing table for IPv4 whose *stride lengths* are as follows: $S_0 = 16$, $S_1 = 8$, and $S_2 = 8$. Assume there are no routes in the routing table, then a route whose destination IP prefix is 10.1.4/22 (say route C) is inserted. The route insertion process in this case is as follows:

1. A level 1 array (say X_1) is allocated and cleared.
2. $X_0[10 \times 256 + 1].pNext = X_1$

3. $X_1[4].pLM = C$
4. $X_1[5..7].pDef = C$

Figure 5 shows level 1 array X_1 after route C is inserted.

Element	Contents		
	pDef	*pLM*	*pNext*
0	NULL	NULL	NULL
1	NULL	NULL	NULL
2	NULL	NULL	NULL
3	NULL	NULL	NULL
4	NULL	C	NULL
5	C	NULL	NULL
6	C	NULL	NULL
7	C	NULL	NULL
8	NULL	NULL	NULL
⋮	⋮	⋮	⋮
255	NULL	NULL	NULL

C: 10.1.4/22

Figure 5: Level 1 Array X_1 After Inserting Route C (10.1.4/22)

Note that $X_1[4].pDef$ does not point to route C. $X_1[4].pDef$ must be one of the following 2 values by definition:

1. $X_1[0].pLM$ unless $X_1[0].pLM$ is NULL
2. NULL if $X_1[0].pLM$ is NULL.

$X_1[4].pDef$ is kept NULL since $X_1[0].pLM$ is NULL. $X_1[4].pDef$ cannot be set to either the value of $X_1[1].pLM$ or the value of $X_1[3].pLM$ since their prefix lengths must be 24. $X_1[4].pDef$ cannot be set to the value of $X_1[2]pLM$, either since its prefix length must be 23 or 24. Actually, there is no need to update $X_l[i].pDef$ when a new route is inserted to $X_l[i].pLM$ (index i of array X at level l). $X_l[i].pDef$ is automatically updated when a less specific route than the route pointed to by $X_l[i].pLM$ is inserted.

Assume now a route whose destination IP prefix is 10.1/20 (say route D) is inserted. The route insertion process in this case is as follows:

1. $X_1[0].pLM = D$
2. $X[1..15].pDef$ are set to D if their value is NULL or the prefix length of the route pointed to by each of them is shorter than 20.

$X_1[5..7].pDef$ do not change since they are pointing to route C and the prefix length of route C is 22. Figure 6 shows the level 1 array after route D is inserted.

Element	Contents		
	pDef	*pLM*	*pNext*
0	NULL	D	NULL
1	D	NULL	NULL
2	D	NULL	NULL
3	D	NULL	NULL
4	D	C	NULL
5	C	NULL	NULL
6	C	NULL	NULL
7	C	NULL	NULL
8	D	NULL	NULL
⋮	⋮	⋮	⋮
15	D	NULL	NULL
16	NULL	NULL	NULL
⋮	⋮	⋮	⋮
255	NULL	NULL	NULL

C: 10.1.4/22
D: 10.1/20

Figure 6: Level 1 Array X_1 After Inserting Route D (10.1/20)

Assume now a route whose destination IP prefix 10.1.4/23 (say route H) is inserted. The *base index* of route H is 4, but $X_1[4].pLM$ already has a pointer to route C. The route insertion process in this case is as follows:

1. $H{\rightarrow}pNext = X_1[4].pLM$
2. $X_1[4].pLM = H$
3. $X_1[5].pDef = H$

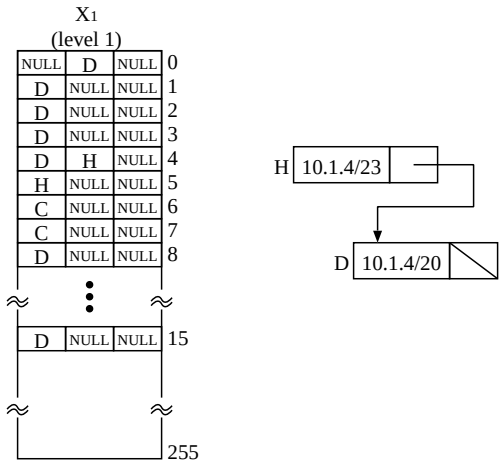
$X_1[5].pDef$ is set to H since the prefix length of route H is longer than that of the route originally stored in $X_1[5].pDef$, which is route C. Figure 7 shows the level 1 array X_1 after route H is added.

Assume now a route whose destination IP prefix is 10/8 (route B) is inserted. In this case, the route insertion process is as follows:

1. $X_0[10 \times 256].pLM = B$
2. $X_0[(10 \times 256 + 1)..(10 \times 256 + 255)].pDef = B$ since their value is NULL.

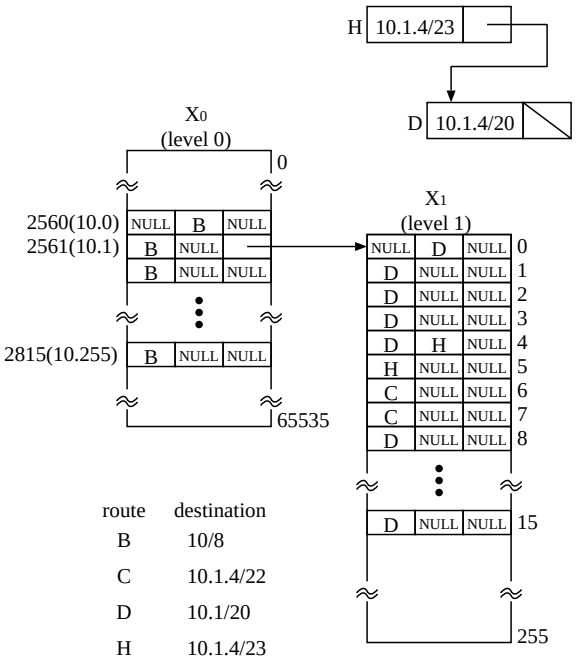
Note that neither level 1 nor level 2 array is accessed. Figure 8 shows the whole SMART after route B is inserted.

Figure 9 shows a pseudo code of the common functions used for the SMART routing table operations. Algorithm 1 shows a rotue insertion pseudo code for SMART.



route	destination
C	10.1.4/22
D	10.1/20
H	10.1/23

Figure 7: Level 1 Array X_1 After Inserting Route H (10.1.4/23)



route	destination
B	10/8
C	10.1.4/22
D	10.1/20
H	10.1.4/23

Figure 8: 4 Routes in SMART

Input: prefix length $plen$
Output: the final level corresponding to ipa
finalLevel($plen$)
(1) Int $level = 0$
(2) Int $len = 0$
(3) **while true**
(4) $len = len + S_{level}$ /* S_{level} is the *stride length* at level $level$ */
(5) **if** $plen \leq len$ **then return** level
(6) $++level$
(7) **endwhile**

Input: IP address $dest$ and corresponding level $level$
Output: *base index* of $dest$ at level $level$
baseIndex($dest$, $level$)
(1) Int $l = 0$
(2) Int $len = 0$
(3) **while** $l \leq level$
(4) $len = len + S_l$
(5) $++l$
(6) **endwhile**
(7) /* W is bit length of IP address */
(8) **return** $(dest >> (W - len))\&(2^{S_{l-1}} - 1)$

Input: prefix length $plen$
Output: number of indices corresponding to $plen$ - 1
getNscan($plen$)
(1) Int $len = 0$
(2) Int $level = 0$
(3) **while true**
(4) $len = len + S_{level}$
(5) **if** $plen \leq len$ **then break**
(6) $++level$
(7) **endwhile**
(8) **return** $2^{(len-plen)} - 1$

Figure 9: Common Functions for SMART Route Operations

Algorithm 1: SMART Route Insertion Algorithm
Input: pointer to a route entry to be inserted $pRtEnt$
Output: **false** if there is the same prefix in table, othewise **true**
insert($pRtEnt$)

```
(1)   /* Allocate new array(s) if necessary */
(2)   Int level = 0
(3)   Int i, nScan, nSkip
(4)   RoutePointer p, q
(5)   Array X = X_0 /* level 0 array */
(6)   while level < finalLevel(pRtEnt→plen)
(7)     i = baseIndex(pRtEnt→dest, level)
(8)     if X[i].pNext == NULL then
(9)       X[i].pNext = New Array
(10)      ++X.nEnt /* number of entries in array X */
(11)    endif
(12)    X = X[i].pNext
(13)    ++level
(14)  endwhile
(15)
(16)  /* Insert pRtEnt to array X */
(17)  ++X.nEnt
(18)  i = baseIndex(pRtEnt→dest, level)
(19)  p = X[i].pLM
(20)  q = NULL
(21)  while p ≠ NULL  and p→plen > pRtEnt→plen
(22)    q = p
(23)    p = p→pNext
(24)  endwhile
(25)  if p ≠ NULL  and p→plen == pRtEnt→plen then
(26)    bcopy(pRtEnt, p, sizeof(*p))
(27)    return false
(28)  endif
(29)  if q == NULL then
(30)    if p == NULL then
(31)      X[i].pLM = pRtEnt
(32)      pRtEnt→pNext = NULL
(33)      goto Update
(34)    else
(35)      q = X[i].pLM
(36)    endif
(37)  endif
(38)  q→pNext = pRtEnt
(39)  pRtEnt→pNext = p
```

```
(30)  /* Update X[i].pDef */
(31)  Update:
(32)  nScan = getNscan(pRtEnt→plen)
(33)  ++i
(34)  while nScan > 0
(35)     if X[i].pDef == NULL  or pRtEnt→plen > X[i].pDef→plen then
(36)        X[i].pDef = pRtEnt
(37)     endif
(38)     /* Skip elements whose element default route is more specific than
         *pRtEnt */
(39)     if X[i].pLM == NULL then
(40)        nSkip = 1
(41)     else
(42)        nSkip = getNscan(X[i].pLM→plen) + 1
(43)     endif
(44)     i = i + nSkip
(45)     nScan = nScan − nSkip
(46)  endwhile
(47)  return true
```

The theoretical maximum number of routing table memory access at route insertion is $2^{S_i-1}-1$ at level i. This value is 255 (128 writes and 127 read) in the case that the *stride length* of an array is 8.

The simplest implementation visits all the array elements associated with the new route and checks the route pointed to by $pDef$, but it is not necessary. When a route poiner is stored in $X_l[i].pLM$, in other words, $X_l[i].pLM$ is not NULL, there is no need to check the value of $pDef$ of the array elements from i to $i + \text{getNscan}(X_l[i].pLM{\rightarrow}plen) - 1$ since the prefix lengths of these routes are always longer than that of the new route. It means that the number of routing table memory accesses can be reduced when routes are inserted in the descendent order of prefix length. This issue is discussed in Section 6.

3.3 Search

Algorithm 2 shows a route lookup pseudo code for SMART.

Algorithm 2: SMART Route Search Algorithm
Input: IP address ipa as a search key
Output: pointer to the longest prefix matching route to ipa, or NULL if there is no such route

```
search(ipa)
(1)   RoutePointer pBest = NULL /* pointer to longest matching route */
(2)   Array X = X_0 /* level 0 array */
(3)   Int level = 0
(4)   Int i = 0
(5)   while true
(6)     i = baseIndex(ipa, level)
(7)     if X[i].pLM ≠ NULL then
(8)       pBest = X[i].pLM
(9)     else if X[i].pDef ≠ NULL then
(10)      pBest = X[i].pDef
(11)    endif
(12)    if X[i].pNext == NULL then return pBest
(13)    X = X[i].pNext
(14)    ++level
(15)  endwhile
```

The following is an example to search the routing table shown in Figure 8 for IP address 10.1.17.1:

1. Variable $pBest$ is initialized to NULL.
2. Array X is set to X_0.
3. Array element $X[10 \times 256 + 1]$ is accessed. $pBest$ is set to B since $X[10 \times 256 + 1].pDef$ is equal to B.
4. Array X is set to $X[10 \times 256 + 1].pNext$.
5. Array element $X[17]$ is accessed. The value of $pBest$ does not change since both $X[17].pLM$ and $X[17].pDef$ are NULL.
6. $pBest$, which is pointing to route B, is returned since $X[17].pNext$ is NULL.

Each time an element of a deeper level array is visited, it is necessary to update local variable $pBest$ with either pLM or $pDef$ of the visited element unless both are NULL as described in Algorithm 2. This is because the route pointed to by one of these pointers becomes the longest matching route when all 3 pointers (pLM, $pDef$, and $pNext$) of the element in the next level array are NULL. It means that the search cost of SMART is 3 times as expensive as the MART in Figure 1 in the worst case since SMART requires 3 memory reads per element visit. In contrast, MART needs one memory read per element visit. The difference of the search cost between the two is however very small in reality. This is because:

1. If one pointer is in the CPU cache or the cache of search hardware, other 2 pointers are also in the cache in the most of cases since these 3 pointers are connected. That is why the possibilities are: 1) all 3 pointers are in the cache, 2) none of 3 pointers is in the cache. In case 1, the cost of 2 more memory accesses is the same as that of 2 more register instructions. In case 2, the cost of 2 more memory accesses is negligible since the cost of loading 3 pointers into the cache is more than 10 times as expensive as that of 2 more cache accesses. That is why the cost of 2 more memory accesses is at the worst 2 more register instructions.

2. It is possible to make it parallel to update $pBest$ and check $X[i].pNext$ in hardware.

3. There is a way to reduce the number of members per element from 3 to 2. This optimization of the basic SMART is discussed in Section 4.

The simulation result of the search performance for MART, MART-CPE, and SMART is shown in section 6.

3.4 Route Deletion

When a route (let us call it route Q) is deleted, it is necessary to update all the array elements that have a pointer to route Q throughout the routing table in MART. In contrast, MART-CPE and SMART require to update only the elements in the same array in which pointers to route Q were stored.

The update consists of two parts. The one is to find the route that replaces route Q. Let us call it route R. Route R must be the second longest-matching route of the destination IP address of route Q. The other is to replace the pointer value in all the necessary array elements as described above. As we saw in section 2, it is expensive to find route R in both MART and MART-CPE since it is necessary to backtrack through the array elements (and arrays in the case of MART). SMART can always find route R with one memory access. Suppose a pointer to route Q is stored in $X[i].pLM$. A pointer to Route R is always stored in either $X[i].pDef$ (in the case $Q{\rightarrow}pNext == NULL$) or $Q{\rightarrow}pNext$ (in the case $Q{\rightarrow}pNext \neq NULL$) because an element default route always points to the second longest-matching route of the associated element by definition.

Algorithm 3 shows a route deletion pseudo code for SMART.

Algorithm 3: SMART Route Deletion Algorithm
Input: Destination IP address ipa and the corresponding prefix length $plen$
Output: **true** if success, otherwise **false**

delete(ipa, $plen$)
(1) Array $X = X_0$ /* level 0 array */
(2) Array $save[0] = X$ /* for freeing arrays */
(3) RoutePointer p, q
(4) Int $level = 0$
(5) Int $i = 0$
(6) Int $nScan$
(7) /* Get the level where the route is to be stored */
(8) **while** $level <$ finalLevel($plen$)
(9) $i =$ baseIndex($ipa, level$)
(10) **if** $X[i].pNext == NULL$ **then break**
(11) $X = X[i].pNext$
(12) $++level$
(13) $save[level] = X$
(14) **endwhile**
(15)
(16) /* Check if there is a matching route */
(17) $i =$ baseIndex($ipa, level$)
(18) $p = X[i].pLM$
(19) **while** $p \neq NULL$
(20) **if** $p \rightarrow plen == plen$ **then goto** DoIt
(21) $q = p$
(22) $p = p \rightarrow pNext$
(23) **endwhile**
(24) **return false**

```
(25)  DoIt:
(26)  /* Remove the matching route */
(27)  if p == X[i].pLM then
(28)      X[i].pLM = p→pNext
(29)  else
(30)      q→pNext = p→pNext
(31)  endif
(32)  /* Get the route for replacing the deleted route */
(33)  if p→pNext == NULL then
(34)      q = X[i].pDef
(35)  else
(36)      q = p→pNext
(37)  endif
(38)
(39)  /* Update X[i].pDef */
(40)  nScan = getNscan(plen)
(41)  ++i
(42)  while nScan > 0
(43)      if X[i].pDef == p then
(44)          X[i].pDef = q
(45)      endif
(46)      /* Skip elements whose element default route is more specific than *p */
(47)      if X[i].pLM == NULL then
(48)          nSkip = 1
(49)      else
(50)          nSkip = getNscan(X[i].pLM→plen) + 1
(51)      endif
(52)      i = i + nSkip
(53)      nScan = nScan − nSkip
(54)  endwhile
(55)
(56)  /* Free route entry and arrays */
(57)  Free p
(58)  while level > 0
(59)      −−X.nEnt
(60)      if X.nEnt ≠ 0 then return true
(61)      Free X
(62)      −−level
(63)      X = save[level]
(64)      X[baseIndex(ipa, level)].pNext = NULL
(65)  endwhile
(66)  −−X.nEnt  /* Don't free level 0 array */
(67)  return true
```

The maximum number of routing table memory access in Algorithm 3 is $2^{S_i-1} - 1$ at level i. This value is 255 (128 writes and 127 read) in the case that the *stride length* of an array is 8.

Same as the case of route insertion, it is not necessary to visit all the array elements associated with the route to be deleted when $X[i].pLM$ is not NULL. It means that the number of routing table memory accesses can be reduced when routes are deleted in the ascendent order of prefix length. This issue is discussed in Section 6.

4 Optimization

The basic SMART routing table operations are described in the previous section. This section discusses an optimization to basic SMART, particularly how to reduce the memory usage and how to increase the search performance.

A basic SMART array element has 3 members (Figure 4). This structure simplifies the routing table operations, but requires a lot of memory. pLM and $pNext$ can be consolidated as shown in Figure 10 by using the least significant 2 bits of $pDef$ as the union identifier of $pNext$.

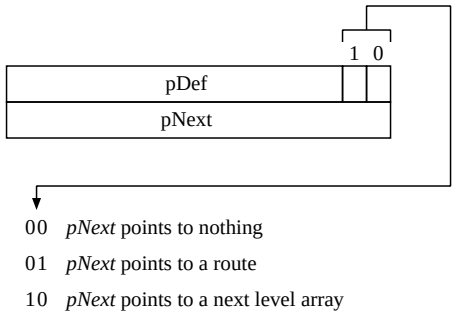


Figure 10: Optimized Smart Array Element

This enhancement saves 33% of memory per element. It also reduces the worst case search cost from 3 to 2 routing table memory accesses per element visit. However, it causes the following 'moving routes' problem. Assume the situation in Figure 11-1. The SMART in Figure 11-1 has only one route, which is route E whose destination IP prefix is 10/15. Assume now a new route (say route F) whose destination IP prefix is 10.0.1/24 is to be inserted to the table. The route inserting function has to:

1. move route E from the level 0 array to the level 1 array as shown Figure 11-2
2. update the element default route.

This means the element default route may be spread over multiple arrays. In this case, the maximum number of routing table access becomes $2^{S_i} - 1$ at level i.

However, it is possible to keep the maximum number of routing table access $2^{S_i-1} - 1$. Note that the worst case at level i happens only when $X_{i-1}[\text{baseIndex}(pNewRoute{\rightarrow}dest, i-1)].pLM$ moves down to $X_i[0].pLM$. Here, $pNewRoute$ is a pointer to the newly inserted route. In this case, $X_i[0].pLM$ can be stored in $X_i[0].pDef$ instead of spreading it to $X_i[1..2^{S_i} - 1].pDef$ since $X_i[0].pDef$ is never used. Let us call this route table default route. The table default route is checked when the search function visits array element j at level i, and both $X_i[j].pDef$ and $X_i[j].pLM$ are equal to NULL. The table default route method keeps the route the update cost $2^{S_i-1} - 1$. The drawback, however, is that the table default route method requires an extra check when a pointer to the element default route is NULL.

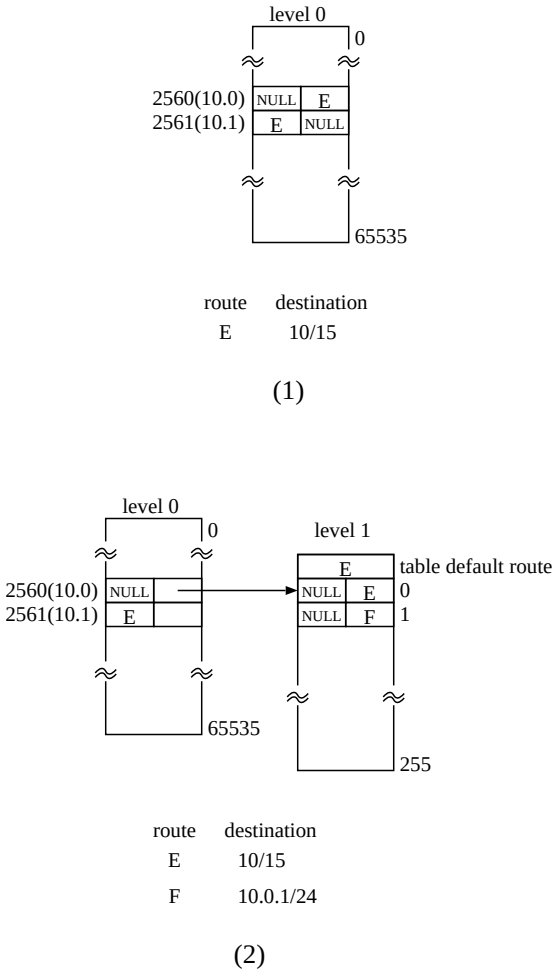


Figure 11: Moving a Route

5 Cost Comparison

Table 1 shows the maximum number of routing table memory accesses (MAX-RTA) of the routing table operations among MART, MART-CPE, and SMART. SMART-TD is an implementation of SMART that uses the table default route method. Let us use W for the length of an address (e.g., 32 for IPv4, 64 for IPv6 [7]), N for the number of sequences (levels), and S_i for the *stride length* at a level i array.

Algorithm	Search	Insertion	Deletion
MTRT	N	$2\prod_{i=0}^{N-1} 2^{S_i}$	$\sum_{i=0}^{N-1} S_i + 2\prod_{i=0}^{N-1} 2^{S_i}$
MTRT-CPE	$2N$	$2(2^{S_i})$	$S_i + 2(2^{S_i})$
SMART	$2N$	$2S_i$	$1 + 2S_i$
SMART-TD	$2N$	S_i	$1 + S_i$

Table 1: Maximum Number of Routing Table Memory Accesses (MAX-RTA)

In Deletion, the left hand side number shows the maximum number of routing table memory

accesses to search for the second longest matching route that replaces the route to be deleted; the right hand side number shows the maximum number of routing table memory accesses to update an array at level i.

5.1 Search

MAX-RTA of MART is N because it requires single memory access per array visit.

MAX-RTA of MART-CPE is $2N$ because it requires 2 memory accesses (reading *pCPE* and *pNext*) per array visit.

MAX-RTA of SMART depends on the implementations. It is $3N$ in the case of 3 members per element, and $2N$ in the case of 2 members per element. However, there is no difference between MART and SMART in reality. This is shown in Section 6.

5.2 Insertion

MAX-RTA of MART is $2S_i^N$ because it spreads a pointer all over the routing table. it is necessary to multiply 2 because MART has to read the pointer of a route first, then it writes a different pointer if necessary.

MAX-RTA of MART-CPE is $2S_i$ because it spreads a pointer within an array.

5.3 Deletion

The paper [5] does not describe how to find the second longest matching route at all. MAX-RTA of the simplest MART is $NS_i + 2S_i^N$.

The paper [6] does not describe how to find the second longest matching route at all. MAX-RTA of the simplest MART-CPE is $S_i + 2S_i$.

MAX-RTA of SMART is $1 + 2S_i$. It becomes $1 + S_i$ when the table default method is used.

6 Simulation and Observed Results

The author performed the following 4 simulations:

- Simulation 1: performance under a route flap
- Simulation 2: deletion performance under steady state
- Simulation 3: table size and performance.
- Simulation 4: random route update vs. sorted route update

MART, MART-CPE, and SMART used in the above simulations have 3 levels and their *stride lengths* are $S_0 = 16$, $S_1 = 8$, and $S_2 = 8$. The simulation is performed on a machine that has the following specifications:

- CPU: AMD Athlon 600MHz
- Memory: 128MB
- OS: Linux 2.2.14
- Compiler: egcs-2.91.66

6.1 Simulation 1: Performance under a Route Flap

This simulation simulated all 3 routing table operations, which are insertion, deletion, and lookup, for the BSD radix implementation [8], a MART implementation, a MART-CPE implementation, and a SMART implementation that has 2 members per array element. The reason the author used sample MART and MART-CPE implementations is that there was no source code of MART and MART-CPE available. MAE-EAST routing table [3] on Aug. 17, 1999 is used as the source of BGP routes. This routing table has 42,366 routes and 95 routes have the prefix length longer than 24. 2,000 random routes whose prefix length is longer than 24 are also created and are inserted to the routing table as the source of IGP (Interior Gateway Protocol) routes. This is because MAE-EAST routing table does not have IGP routes whose prefix lengths are usually longer than 24. It is important to include the IGP routes whose prefix length is longer than 24 because it significantly increases the number of both level 1 array and level 2 array (see Table 6). One should not ignore IGP routes at simulation because it is common that ISP (Internet Service Provider) routers have both BGP and IGP routes in their routing tables in the real world, and again, the prefix lengths of most of the IGP routes are longer than 24. Table 2 shows the prefix length distribution of the routing table used for the simulation.

prefix length	number of routes
8	20
9	3
10	3
11	9
12	21
13	42
14	110
15	184
16	4869

prefix length	number of routes
17	496
18	1081
19	3285
20	1723
21	2082
22	2790
23	3582
24	21971

prefix length	number of routes
25	435
26	438
27	407
28	406
29	406
30	3

Total Number of Routes: 44,366

Table 2: Prefix Length Distribution

The simulation process:

1. randomly inserts 44,366 routes to the routing table
2. looks up 100,000 random IP addresses in the routing table
3. randomly deletes all 44,366 routes from the routing table
4. repeats 1 to 3 for 100 times.

The simulation inserts, looks up, and deletes routes randomly so that any order dependent effect is avoided.

Table 3 shows the simulation result of the tree routing table implementations.

Table 4 shows the distribution of the number of backtrack to find the second longest matching route in MART-CPE at deletion.

	search (μs/route)	insert (μs/route)	delete (μs/route)	memory use (MB)
BSD Radix	2.00	4.06	3.61	5.54
MART	0.20	2.03	3.83	9.98
MART-CPE	0.20	1.13	2.03	17.42
SMART	0.20	1.35	1.35	17.42
SMART-TD	0.20	1.13	1.13	17.42

- The values in μs are the average of 44,366 random insertion/deletion and 100,000 random IP addresses lookup.
- The memory use is when there are 44,366 routes in the table.
- The simple backtracking method is used in MART and MART-CPE for deletion.
- SMART-TD uses the table default route method.

Table 3: Performance of BSD Radix, MART, MART-CPE, and SMART

number of backtracks			number of routes	%
		0	2,051	4.6
		1	32,581	73.4
2	..	9	7,371	16.6
10	..	19	584	1.3
10	..	39	778	1.8
40	..	59	120	0.3
60	..	99	535	1.2
100	..	255	346	0.8

Table 4: Backtracking Number Distribution (MART-CPE)

6.2 Simulation 2: Deletion Performance under Steady State

The purpose of this simulation is to compare the deletion performance between MART-CPE and SMART under the condition that not many routes are deleted at one time. The simulation process:

1. randomly inserts 44,366 routes to the routing table
2. looks up 1,000 random IP addresses in the routing table
3. randomly deletes 10 routes from the routing table
4. repeats 2 and 3 for 10 times.

Table 6.2 shows the result of Simulation 2.

	delete (μs/route)
MART	2.35
SMART-TD	1.81

Table 5: Route Deletion Performance under Steady State

6.3 Simulation 3: Table Size and Performance

Table 6 shows the performance difference between two routing tables; the one (MAE-EAST) is the pure MAE-EAST routing table that has only 95 routes whose prefix length is longer than 24, the other (MAE-EAST+) is a table which has all the MAE-EAST routes plus 2,000 randomly created routes whose prefix length is longer than 24.

	level 1 array	level 2 array	search (μs/route)	insert (μs/route)	delete (μs/route)
MAE-EAST	3,441	64	0.20	0.94	0.94
MAE-EAST+	5,299	2,064	0.20	1.13	1.13
diff.(%)	53.99	3,125	0.00	20.21	20.21

- MAE-EAST has 42,366 routes and 95 routes' prefix length is longer than 24
- MAE-EAST+ has 2,000 more routes whose prefix length is longer than 24 in addition to the MAE-EAST routing table

Table 6: Table Size and Performance

6.4 Simulation 4: Random Route Update vs. Sorted Route Update

Table 7 shows the performance difference between the following two cases:

1. 44,366 routes are inserted and deleted randomly
2. 44,366 routes are inserted according to the descending order of the prefix length and deleted according to the ascending order of the prefix length.

The IP addresses that have the same prefix length are inserted and deleted randomly. Search performance is measured with 100,000 random IP addresses lookup.

	search (μs/route)	insert (μs/route)	delete (μs/route)
random	0.20	1.13	1.13
sorted	0.20	0.68	0.68

Table 7: Update Order and Performance

6.5 Observed Results

The simulation results show the following things:

1. SMART has 79.6% as fast as MART-CPE in deletion under route flap even though 78% of the second longest matching routes are found with 0 or 1 backtracking. This is because:
   (a) 'while' loop overhead is not negligible

(b) many memory accesses happen even though the percentage of long backtracking is low (about 22.0%).

This result shows that SMART is much better than MART-CPE when a route flap happens although SMART has few advantages against MART-CPE in the steady state.

2. SMART-TE has 29.8% as fast as MART-CPE in deletion under steady state. It means SMART deletion performance is always better than that of MART-CPE.

3. SMART is more than 10 times in search, more than 3 times in insertion and deletion as fast as BSD radix when the table default method is applied. On the other hand, SMART uses more than 3 times as much memory as BSD radix.

4. There is no difference in search performance among MART, MART-CPE, SMART, and SMART with the table default route method. This result suggests that all the saved element default route and the table default route are in the cache. It is quite possible because the Athlon CPU has 64K byte L1 data cache.

5. SMART improves route insertion and deletion performance compared to MART. SMART is almost twice in route insertion, more than 3 times in deletion as fast as the traditional MART.

6. The route insertion and deletion performance improves 16% when the table default method is used. This result suggests that the cost of accessing the whole array affects the performance. Actually, the number of routes whose prefix length is 24 is large (21,971). These routes move to the level 2 arrays when more specific routes exist and it is necessary to access the whole array elements unless the table default route is applied.

7. Neither the number of routes nor the number of deep level arrays seriously affects the route update performance. The number of routes increased almost 54%, the number of level 2 array increased 32 times in Table 6. However, the route update cost change is less than 17%. This is an expected result of the SMART route update algorithm.

8. When a route flap happens, the performance of both insertion and deletion increases about 66% in the case these routes are sorted in SMART. Modern routers usually have two routing tables. One is owned by the controller that handles routing protocols and the other is owned by the packet forwarder. The controller calculates routes and downloads them to the forwarder's routing table. The update performance of the forwarder increases about 66% when the controller sorts routes according to the prefix length and downloads them to the forwarder. It is inexpensive to sort routes according to only the prefix length.

The simulation result shows that the performance of SMART is pretty good for all 3 routing table operations. Furthermore, it is easy to implement the SMART search in hardware and make it pipelined. The search performance becomes single memory access speed in this case.

7 Conclusions

It is important to keep the route update cost low in order to quickly recover from route flaps. The number of IGP routes whose prefix length is longer than 24 is not negligible, either. The route

update cost of simple MART is 32M and more routing table memory accesses in the worst case. The author presented a new multi-array routing table called SMART (Smart Multi-Array Routing Table) whose route update cost is always less than $2^{S_i-1} - 1$ routing table memory accesses. SMART does not spread route pointers all over the routing table. Instead, each SMART array element has an element default route, which is the second longest-matching route of the associated array element. In theory, the element default route affects the search performance, but it does not in reality. The SMART search cost becomes one memory access when it is implemented in hardware with pipelining. The author showed that SMART is 79.6% faster than MART-CPE in deletion and the SMART search cost is practically the same as that of simple MART by simulation. The author also showed that SMART is 10 times in search, more than 3 times in insertion and deletion as fast as the BSD radix tree routing table by simulation.

Acknowledgments

The author would like to thank Tom Herbert, Steve Reseigh, Paul Ziemba, Kevin Williams, and Donald Knuth for useful comments on a draft of this paper. The author would like to thank MAYAN Networks for giving him an opportunity to write this paper. The author would also thank Hiroki Nakano who helped to port the BSD radix code to a user program.

References

[1] Tony Bates, *Routing Table History*, http://www.employees.org:80/∼tbates/cidr.plot.html

[2] V. Fuller, T. Li, J. Yu, and K. Varadhan, *Classless Inter-Domain Routing (CIDR)*, RFC1519, September 1993.

[3] Merit Networks, Inc., *Internet Routing Trends* http://www.telstra.net/ops/bgptable.html

[4] C. Labovitz, R. Malan, F. Jahanian, *Internet Routing Stability*, Proceedings of ACM SIGCOMM, Sept. 1997.

[5] Pankaj Gupta, Steven Lin, and Nick McKeown, *Routing Lookups in Hardware at Memory Access Speeds*, Proceedings of Infocom, April 1998.

[6] V. Srinivasan and George Varghese, *Faster IP Lookups using Controlled Prefix Expansion*, Proceedings of ACM Sigmetrics, Sep 98 and ACM TOCS 99.

[7] R. Hinden, M. O'Dell, S. Deering, *An IPv6 Aggregatable Global Unicast Address Format*, RFC2374, July 1998.

[8] FreeBSD 2.2.2, /usr/src/sys/net/radix.[ch]