ISSN 1975-8359(Print) / ISSN 2287-4364(Online)
The Transactions of the Korean Institute of Electrical Engineers Vol. 65, No. 12, pp. 2061~2068, 2016
http://dx.doi.org/10.5370/KIEE.2016.65.12.2061

벡터 블룸 필터를 사용한 IP 주소 검색 알고리즘

IP Address Lookup Algorithm Using a Vectored Bloom Filter

변 하 영* · 임 혜 숙†
(Hayoung Byun · Hyesook Lim)

Abstract - A Bloom filter is a space-efficient data structure popularly applied in many network algorithms. This paper proposes a vectored Bloom filter to provide a high-speed Internet protocol (IP) address lookup. While each hash index for a Bloom filter indicates one bit, which is used to identify the membership of the input, each index of the proposed vectored Bloom filter indicates a vector which is used to represent the membership and the output port for the input. Hence the proposed Bloom filter can complete the IP address lookup without accessing an off-chip hash table for most cases. Simulation results show that with a reasonable sized Bloom filter that can be stored using an on-chip memory, an IP address lookup can be performed with less than 0.0003 off-chip accesses on average in our proposed architecture.

Key Words : Bloom filter, IP address lookup, Vectored bloom filter

1. Introduction

An IP address consists of a network part (called a prefix) and a host part. The network part identifies a group of hosts included in a network and the host part identifies a specific host [1]. Under a class-based addressing scheme, the length of the network part was fixed as 8, 16, or 24 bits. An exact matching operation was performed for an IP address lookup to forward each packet toward a final destination at Internet routers. However, excessive prefix waste was caused by the inflexibility in network sizes under the class-based addressing scheme. A new addressing scheme called classless inter-domain routing (CIDR) is currently being used. The CIDR allows variable-length prefixes, and the Internet routers use the longest prefix of all matching prefixes as the best matching prefix (BMP) to forward each input packet to the most specific network [2-5].

Today, as the speed of Internet traffic continues to increase exponentially, users utilize a variety of network applications demanding real-time services. Hence, the IP address lookup has become one of the most challenging functionalities that need to be performed at wire-speed. Various IP address lookup algorithms have been studied such as trie-based [4-5], hashing-based [6], and Bloom filter-based algorithms [7-9]. Because of their sizes, the trie and the hash table are usually stored using off-chip memories, and IP address lookup procedures are completed through off-chip memory accesses. However, an access to an off-chip memory is 10 to 20 times slower than access to an on-chip memory [10]. An on-chip Bloom filter has been used to reduce the number of off-chip memory accesses [7-9].

In this paper, we propose a new Bloom filter-based IP address lookup algorithm using a vectored Bloom filter (VBF).

The remainder of this paper is organized as follows. Section 2 describes related works and Section 3 introduces our proposed IP address lookup algorithm. Section 4 evaluates and compares the performance of our proposed algorithm with previous BF-based algorithms and Section 5 concludes the paper.

2. Related Works

2.1 Bloom Filter

A Bloom filter [11] is a bit-vector-based data structure used to represent a data set and to answer membership queries. Bloom filters have been popularly used in many network algorithms because of their space efficiency [12]. A Bloom filter involves two operations: programming and querying.

† Corresponding Author : Dept. of Electronic and Electrical Engineering, Ewha Womans University, Korea.
E-mail: hlim@ewha.ac.kr
* Dept. of Electronic and Electrical Engineering, Ewha Womans University, Korea.
Received : February 11, 2016; Accepted : November 7, 2016

A Bloom filter is an array of m bits, which are initially set to 0. In programming, every element in a set $S = \{x_1, x_2, \cdots, x_n\}$ is programmed to the Bloom filter using k independent hash functions, each of which is used to map an element to a random number with a range $\{1, \cdots, m\}$. To insert an element, the k bits corresponding to the k hash indices are set to 1. For n given elements, the optimal number of hash functions is $k = (m/n)\ ln2$.

Using the same k hash functions as those used in the programming, querying is performed to check whether an input is the member of the set. In querying an input, if any of the k bits are 0, the input is definitely not a member of set S, and it is defined as a *negative*. If all the k bits are 1, the input is identified as a member of set S, and it is defined as a *positive*.

A Bloom filter can produce false positives due to hash collision; an input $y \notin S$ can have k hash indices corresponding to k bits set to 1. False positives can be improved by increasing the size of the Bloom filter, but cannot be completely eliminated. However, the negative results of the Bloom filter are always true.

2.2 Bloom Filter-Based IP Address Lookup Algorithms

Dharmapurikar et al. proposed an IP address lookup algorithm by employing Bloom filters [7]. Their structure consists of W Bloom filters which are queried in parallel, where W is the number of different prefix lengths. For the lengths with positive results, an off-chip hash table is probed and returns the output port corresponding to the longest matching prefix.

Lim et al. proposed to add a Bloom filter to trie-based algorithms [9]. In performing a binary search on trie levels [13-14], an on-chip Bloom filter is used to reduce the number of hash table accesses by producing negative results for non-existing nodes.

In the Bloom filter chaining approach proposed by Mun et al. [15], an on-chip Bloom filter is programmed for each node in a trie and is sequentially queried as increasing the trie level. For a specific length of a given input, if a negative result is produced, it means that no node appears at the current level and at longer levels for the path of the input. Hence, an off-chip hash table is accessed for the last positive level. False positives of the Bloom filter can cause back-tracking to a shorter level.

3. Proposed Algorithm

The proposed vectored Bloom filter (VBF) is an m multi-bit array, which contains an output port in each vector. Our proposed structure completes the IP address lookups by querying only the VBF without accessing the off-chip hash table. The hash table is constructed using an off-chip memory for the case where an output port is not determined through the VBF. Fig. 1 shows the vectored Bloom filter structure.

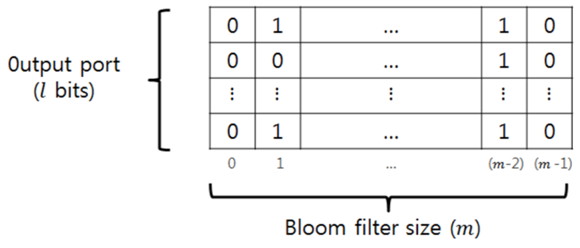


Fig. 1 Vectored Bloom filter structure

3.1 Basic Vectored Bloom Filter Algorithm

As shown in Fig. 1, each hash index for the VBF points to l bits which represent an output port. In our proposed approach, if all of the l bits for any single vector are 0, it is defined as a negative, and it means that the vector is not programed by any prefix. If all of the l bits for a single vector are 1, it is defined as a *conflict*, and it means that the vector is programed by two or more different prefixes. Hence, an l-bit vector can represent up to 2^l-2 output ports.

In the construction procedure, every prefix in a routing table is programmed into the VBF and is stored in the hash table. Algorithm 1 describes the details in programming the basic VBF. For a prefix x which has output port *x.port*, k hash indices are obtained. The k vectors corresponding to the hash indices are written by *x.port*. If any of the vectors already have an output port other than *x.port*, the vector is written by all 1s to represent the *conflict*. This procedure is repeated for every prefix included in the routing table.

Algorithm 2 describes the details in querying the basic

Algorithm 1 Basic VBF Programming Procedure

```
Function programVBF(x)
  for (i = 1 → k)
    if (BF[h_i(x)] == NULL) then
      BF[h_i(x)] ← x.port;
    else if (BF[h_i(x)] != x.port) then
      for (j = 0 → l-1)
        BF[h_i(x)][j] ← 1;   // conflict
      end for
    end if
  end for
end Function
```

Algorithm 2 Basic VBF Querying Procedure

```
Function queryVBF(y)
   check ← BF[h_0(y)] & BF[h_1(y)] & … & BF[h_{k-1}(y)];
   counter ← 0;
   if (all bits in check is 0) then
      return 0;  //negative
   else if (all bits in check is 1) then
      return -1;  //conflict
   else
      for (i = 1 → k)
         if (check == BF[h_i(y)]) then
            counter ← counter + 1;
         else if (all bits in BF[h_i(y)] is 1) then
               counter ← counter + 1;
         end if
      end for
      if (counter < k) then
         return 0;  //   negative
      else
         outPort ← convert_decimal(check);
         return outPort;   // positive
      end if
   end if
end Function
```

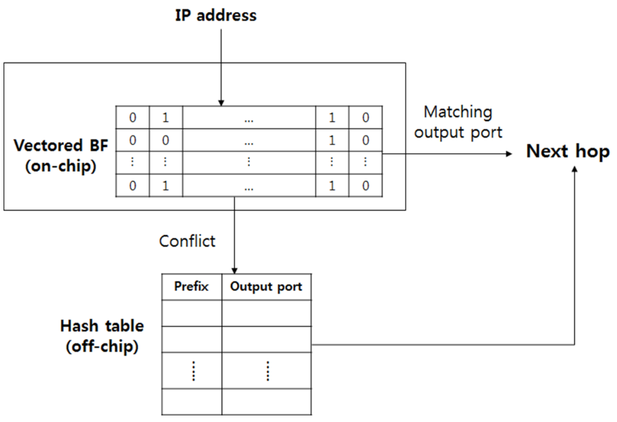


Fig. 2 The search procedure of proposed algorithms

VBF. We use ‘AND’ operation in this algorithm. For example, suppose we use three 4-bit vectors; $k = 3$ and $l = 4$. If the result of ‘AND’ operation with 3 vectors is 0000, it indicates a negative result. If the result is 1111, it indicates a *conflict* result. If the result is neither 0000 nor 1111, we should check whether it is a positive result. For example, if the three 4-bit vectors are 1111, 0001, and 0011, it is clearly a negative result because two different output ports are programmed to the vectors. We implemented this function as follows. Since the result of AND operation of these three vectors is 0001, we count the number of 0001 and 1111. If the counting result is less than k, which is 2 in this example, it is a negative. If it is the same as k, it returns the output port.

The search procedure for the proposed IP address lookup algorithm is shown in Fig. 2 and Algorithm 3. For a given

Algorithm 3 VBF Search Procedure

```
Function Search(DstAddr)
   for (length = longestLen → shortestLen)
      if (queryVBF(DstAddr.length) == -1) then
         // conflict
         BMPport ← searchHT(DstAddr.length);
         if (BMPport != NULL) then
            break;  // no more Bloom filter access
         end if
      else if (queryVBF(DstAddr.length) == 0) then
         continue;  // negative
      else            // positive
         BMPport ← queryVBF(DstAddr.length);
         break;
      end if
   end for
   return BMPport;
end Function
```

input address, starting from the longest length of prefixes, the VBF is queried and k vectors are obtained. If the VBF querying returns the *conflict*, whether the given input has a matching prefix in the current length is not determined. Hence the off-chip hash table should be accessed. If the hash table returns an output port for the current length, the search procedure is finished. Otherwise, the search procedure continues to a shorter length.

3.2 Refined Vectored Bloom Filter Algorithm

A drawback of the basic VBF algorithm described in 3.1 is that a vector becomes useless when being programmed with the *conflict*. The proposed algorithm is refined to utilize the information included in the *conflict* vectors. In the refined structure, one bit is allocated for each output port. For each of the k vectors obtained for a prefix, the bit location of the output port corresponding to this prefix is set. The structure of this refined algorithm is the same as Fig. 1, an l-bit vector can represent l different output ports in the refined structure, while it can represent 2^l-2 output ports in the basic structure.

Algorithm 4 describes programming procedure of the refined VBF. In programming a prefix, each bit of the k vectors, which corresponds to the output port of the prefix, is changed to 1. Hence, even though different ports have already been programmed in a vector, the vector is not required to indicate the *conflict*. Note that each row of the

Algorithm 4 Refined VBF Programming Procedure

```
Function programVBF(x)
   for (i = 1 → k)
      BF[h_i(x)][ x.port-1] ← 1;
   end for
end Function
```

Algorithm 5 Refined VBF Querying Procedure

```
Function queryVBF(y)
  check ← BF[h_0(y)] & BF[h_1(y)] & … & BF[h_{k-1}(y)];
  counter ← 0;
  for (i = 0 → l-1)
    if (check[i] == true) then
      outPort ← i + 1;
      counter ← counter + 1;
      if (counter > 1) then
        break;
      end if
    end if
  end for
  if (counter == 0) then
    return 0; // negative
  else if (counter > 1) then
    return -1; //conflict
  else if (counter == 1) the
    return outPort; // positive
  end if
end Function
```

refined VBF represents the membership of the prefixes having the same output port. Assuming that prefixes are evenly distributed to each output port, each row of the refined VBF is programmed by n/l prefixes.

Algorithm 5 describes the querying procedure of the refined VBF. If none of the vectors obtained for a specific length of the input have a common bit set to 1, it is a negative result. If all vectors have two or more common bits set, it is the *conflict* case. If a specific bit location of all vectors is set, it is the output port of the matching prefix. We also use 'AND' operation to implement this. Since the set positions in the result of 'AND' operation denote programmed ports, we count the number of 1s in the result. For example, suppose we use three 4-bit vectors; k = 3 and l = 4. If the result of 'AND' operation is 0000, it indicates a negative result. If the result of 'AND' operation has a single bit set, it indicates a positive result, in which the set position indicates the output port. The *conflict* is indicated when the number of 1s in the result of 'AND' operation is more than 1.

The search procedure for the refined algorithm is the same as the basic algorithm shown in Algorithm 3.

4. Performance Evaluation

Performance evaluation was carried out with C language using routing tables downloaded from five backbone routers. The hash function used for our simulation is a 64-bit cyclic redundancy check (CRC) generator. Multiple hash indices with variable lengths are obtained by combining a variable number of bits of the CRC code. Assuming that the number of output ports is 32, 6 bits and 32 bits are allocated for each vector of the Bloom filter in the basic structure and the refined structure, respectively.

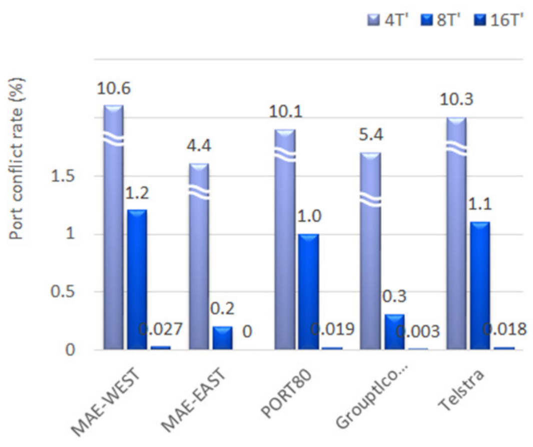


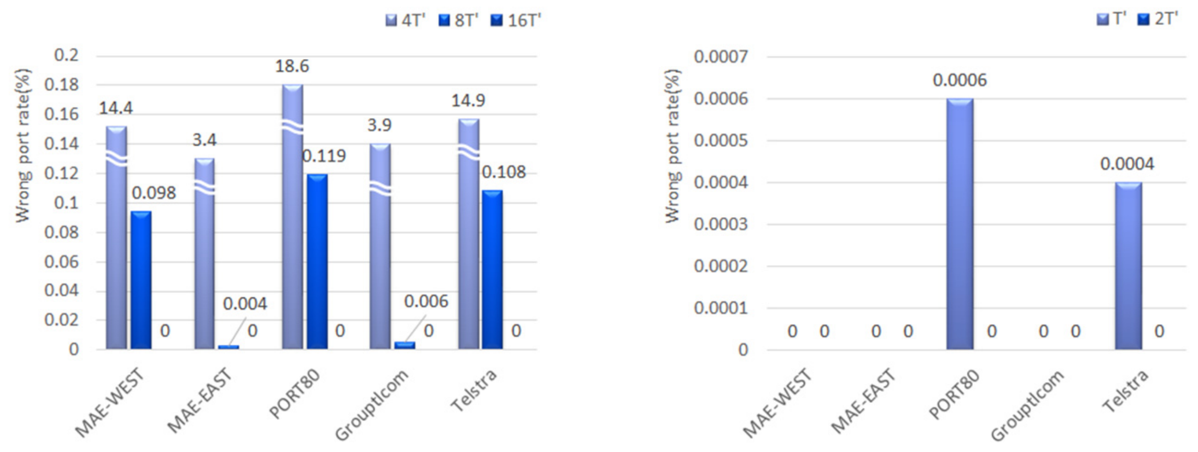
Fig. 3 *Port conflict rate* according to Bloom filter sizes (Proposed Basic Structure)

The *port conflict rate* is defined as the ratio of inputs, in which the output port is not identified by the Bloom filter querying because of vector *conflict*. Hence it indicates the ratio of inputs that should access the off-chip hash table. The *wrong port rate* is defined as the ratio of inputs, in which the Bloom filter provides a wrong result because of a false positive. For the number of elements T programmed to a Bloom filter, the size of a Bloom filter $m = aT'$, where $T' = 2^{\lceil \log_2 T \rceil}$ and a = 4, 8, and 16. The optimal number of hash functions is $k = (m/T')\ ln2$ in the basic structure, while the optimal number of hash functions is $k = (m/R')\ ln2$ in refined structure, where R is the number of prefixes belong to a single port, and $R' = 2^{\lceil \log_2 R \rceil}$. If we assume that prefixes are evenly distributed to each output port, $R = T/l$.

Fig. 3 shows the *port conflict rate* according to Bloom filter sizes. Since the refined structure did not cause any port conflict, we do not show the *port conflict rate* for the refined structure in Fig. 3. Note that the *port conflict rate* is close to 0% for $16T'$.

Fig. 4 shows the *wrong port rate* for the basic structure where a = 4, 8, and 16, and for the refined structure where a=1 and 2. Note that the *wrong port rate* converges to 0% for $16T'$ in the basic structure and for $2T'$ in the refined structure.

Table 1 compares on-chip memory requirements for Bloom filters. The WBSL-BF and LBSL-BF are algorithms proposed in [9], in which a Bloom filter is added to Waldvogel's binary search on trie levels (WBSL) [13] and to Lim's binary search



(a) Proposed basic structure (b) Proposed refined structure

Fig. 4 *Wrong port rate* according to Bloom filter sizes

Table 1 Comparison of On-Chip Memory Requirement for Bloom Filters
(N = number of prefixes, T = number of elements programmed to the Bloom filter, M_b = memory requirement of the Bloom filter)

Routing Data(N)	a	WBSL-BF		LBSL-BF		Chaining-PC		Chaining-LP		Prop-Basic		a	Prop-Ref	
		T	M_b(kB)	T	M_b(kB)	T	M_b(kB)	T	M_b(kB)	T	M_b(kB)		T	M_b(kB)
MAE-WEST (14553)	4	76708	64	82156	64	76708	64	82156	64	14553	48	1	14553	64
	8		128		128		128		128		96	2		128
	16		256		256		256		256		192	-		-
MAE-EAST (39464)	4	172418	128	191757	128	172418	128	191757	128	39464	192	1	39464	256
	8		256		256		256		256		384	2		512
	16		512		512		512		512		768	-		-
PORT80 (112310)	4	225050	128	299899	256	225050	128	299899	256	112310	384	1	112310	512
	8		256		512		256		512		768	2		1024
	16		512		1024		512		1024		1536	-		-
Grouptlcom (170601)	4	314986	256	411122	256	314986	256	411122	256	170601	768	1	170601	1024
	8		512		512		512		512		1536	2		2048
	16		1024		1024		1024		1024		3072	-		-
Telstra (227223)	4	452732	256	576370	512	452732	256	576370	512	227223	768	1	227223	1024
	8		512		1024		512		1024		1536	2		2048
	16		1024		2048		1024		2048		3072	-		-

on trie levels (LBSL) [14], respectively. The Chaining-PC and the Chaining-LP are the algorithms proposed in [15], which perform linear querying to the Bloom filter programmed for nodes in a BMP pre-computed (PC) trie and a leaf-pushing (LP) trie, respectively. In this table, N is the number of prefixes, T is the number of elements programmed to the Bloom filter, and M_b is the memory size in Kilobytes (KB) for the size of a Bloom filter $m = aT'$, where $T' = 2^{\lceil \log_2 T \rceil}$ and a = 4, 8, and 16 in basic structure and a = 1 and 2 in refined structure.

While every node in a trie is programmed to the Bloom filter for other algorithms, prefixes only are programmed in our proposed algorithms. Since each location of the Bloom filter is a vector storing an output port, the memory requirement for a Bloom filter is greater in our proposed algorithm. Since the *port conflict rate* and the *wrong port rate* are both 0% for $2T'$ in our refined algorithm, the simulation from $4T'$ is not performed. In our proposed algorithms, the maximum memory requirement is 3MB, and hence the Bloom filter can be accommodated in an on-chip

Table 2 Comparison of Off-Chip Memory Requirements for Hash Tables
(N = number of prefixes, N_h = number of hash table entries, M_h = memory requirement of the hash table)

Routing Data (N)	WBSL-BF		LBSL-BF		Chaining-PC		Chaining-LP		Prop-Basic		Prop-Ref	
	N_h	M_h(kB)	N_h	M_h(kB)	N_h	M_h(kB)	N_h	M_h(kB)	N_h	M_h(kB)	N_h	M_h(kB)
MAE-WEST (14553)	76708	449.46	82156	481.38	62763	367.75	62685	367.29	14553	85.27	14553	85.27
MAE-EAST (39464)	172418	1010.26	191757	1123.58	134766	789.64	134139	785.97	39464	231.23	39464	231.23
PORT80 (112310)	225050	1318.65	299899	1757.22	179069	1049.23	154767	906.84	112310	658.07	112310	658.07
Grouptlcom (170601)	314986	1845.62	411122	2408.92	246544	1444.59	208168	1219.73	170601	999.62	170601	999.62
Telstra (227223)	452732	2652.73	576370	3377.17	329929	1933.18	290768	1703.72	227223	1331.38	227223	1331.38

Table 3 Comparison of the number of On-Chip Bloom Filter Querying
(N = number of prefixes, A_b = average number of Bloom filter accesses, W_b = worst-case number of Bloom filter accesses)

Routing Data(N)	a	WBSL-BF		LBSL-BF		Chaining-PC		Chaining-LP		Prop-Basic		a	Prop-Ref	
		A_b	W_b	A_b	W_b	A_b	W_b	A_b	W_b	A_b	W_b		A_b	W_b
MAE-WEST (14553)	4	4.36	5	4.89	5	15.91	22	14.94	22	7.42	22	1	8.16	22
	8					15.87		14.89		8.16		2	8.16	
	16					15.87		14.89		8.16		-	-	
MAE-EAST (39464)	4	4.33	5	4.75	5	16.24	22	13.28	22	7.72	22	1	7.88	22
	8					16.17		13.20		7.88		2	7.88	
	16					16.17		13.20		7.88		-	-	
PORT80 (112310)	4	4.72	5	4.71	5	15.26	25	15.39	25	10.71	25	1	11.96	25
	8					15.16		15.35		11.95		2	11.96	
	16					15.15		15.34		11.96		-	-	
Grouptlcom (170601)	4	4.67	5	4.57	5	15.36	20	15.59	20	6.61	20	1	6.77	20
	8					15.32		15.50		6.77		2	6.77	
	16					15.31		15.49		6.77		-	-	
Telstra (227223)	4	4.78	5	4.77	5	17.58	25	17.65	25	8.47	25	1	9.43	25
	8					17.48		17.61		9.43		2	9.43	
	16					17.47		17.61		9.43		-	-	

memory.

Table 2 shows the off-chip memory requirement for a hash table, where N_h is the number of hash table entries and M_h is the memory requirement of the hash table in Kilobytes (KB). The memory requirements for hash tables in our proposed algorithms are estimated as $6N$ bytes, where N is the number of prefixes in a routing table. Our proposed algorithms require the smallest off-chip memories because prefixes only are stored in the hash tables.

Table 3 shows the average number and the worst-case number of Bloom filter accesses stored in the on-chip memory. The A_b and W_b are the average number and the worst-case number of Bloom filter accesses, respectively. The BSL algorithms provide search performance of O($logW$), where the length of IP address W is 32 for IPv4. BF-chaining and our proposed algorithms are based on linear search on the length, and hence the search performance is O(W). However, our algorithms have better performance than the BF-chaining, because our approach proceeds from the longest to the shortest length, while the BF-chaining proceeds from the shortest to the longest length. In case of $16T'$ in the basic algorithm and $1T'$ in the refined algorithm, an IP lookup is performed only through the VBF queries of 6.77 to 11.96 in average. The W_b of our algorithms is less than 32 because the Bloom filter is queried for valid lengths which include at least one prefixes.

The IP address lookup performance mainly depends on the number of off-chip table accesses. Table 4 shows the average number and the worst-case number of hash table accesses stored in the off-chip memory. The A_h and W_h are the average number and the worst-case number of hash table accesses, respectively. All previous algorithms should access the hash table at least once to obtain a matching output port. However, in our proposed algorithm, the hash table is accessed only when port conflicts occur. Hence, the average number becomes 0 as the Bloom filter size increases. It is shown that the proposed refined structure does not require off-chip hash table accesses at all.

5. Conclusion

In this paper, we proposed a new IP address lookup algorithm using a vectored Bloom filter. The vectored Bloom filter can answer membership queries with output ports. The proposed approach improves the address lookup performance by decreasing the number of off-chip memory accesses since the off-chip hash table is accessed only when port conflicts occur. The simulation result showed that the *port conflict rate* and the *wrong port rate* both converge to 0% as the size of the vectored Bloom filter increases. Hence, the proposed algorithm can provide the IP address lookup without an off-chip hash table access.

감사의 글

본 연구는 한국연구재단(NRF)의 중견연구자지원사업 도약과제-미래유망(2014R1A2A1A11051762), 도약과제-성과확산(2015R1A2A1A15054081)과 정보통신산업진흥원의 대학 IT연구센터 지원 사업(IITP-2016-H8501-16-1007)의 연구비 지원으로 수행한 연구로써, 관계부처에 감사드립니다.

References

[1] H. J. Chao, “Next Generation Routers,” *Proc. IEEE*, Vol. 90, No. 9, pp. 1518-1588, Sep. 2002.

[2] S. Fuller, T. Li, J. Yu, and K. Varadhan, “Classless Inter-Domain Routing(CIDR): An Address Assignment and Aggregation Strategy,” *RFC 1519*, Sep.1993.

[3] M. A. Ruiz-Sanchez, E. M. Biersack and W. Dabbous, “Survey and Taxonomy of IP Lookup Algorithms”, *IEEE Networks*, Vol. 15, No. 2, pp. 8-23, Mar./Apr. 2001.

[4] H. Lim and N. Lee, “Survey and Proposal on Binary Search Algorithms for Longest Prefix Match,” *IEEE Communications Surverys and Tutorials*, Vol. 14, No. 3, pp. 681-697, Third Quarters, 2012.

[5] T. Yand, G. Xie, Y. Li, Q. Fu, A. Liu, Q. Li, and L. Mathy, “Guarantee IP Lookup Performance with FIB Explosion,” *ACM Sigcomm*, pp. 39-50, 2014.

[6] P. Gupta, S. Lin, and N. Mckeown, “Routing Lookups in Hardware at Memory Access Speed.” *IEEE INFOCOM*, pp.1240-1247, 1998.

[7] S. Dharmapurikar, P. Krishnamurthy, and D. Taylor, “Longest Prefix Matching Using Bloom Filters,” *IEEE/ACM Trans. Networking*, Vol. 14, No. 2, pp. 397-409, Feb. 2006.

[8] Y. Wang, T. Pan, Z. Mi, H. Dai, X. Guo, T. Zhang, B. Liu, and Q. Dong, “NameFilter: Achieving Fast Name Lookup with Low Memory Cost via Applying Two-Stage Bloom Filters,” *in Proceedings of the IEEE INFOCOM’13*, pp. 93-99, 2013.

[9] H. Lim, K. Lim, N. Lee, and K. Park, “On Adding Bloom Filters to Longest Prefix Matching Algorithms,” *IEEE Trans. Computers*, Vol. 63, No. 2, pp. 411-423, Feb. 2014.

[10] P. Panda, N. Dutt, and A. Nicolau, “On-Chip vs. Off-Chip Memory: The Data Partitioning Problem in Embedded Processor-Based Systems,” *ACM Transactions on Design Automation of Electronics Systems*, Vol. 5, No. 3, pp. 682-704, July 2000.

[11] B. H. Bloom, “Space/Time Trade-offs in Hash Coding with Allowable Errors,” *Communications of the ACM*, Vol. 13, No. 7, pp. 422-426, 1970.

[12] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, “Theory and Practice of Bloom Filters for Distributed Systems,” *IEEE Communications Surveys and Tutorials*, Vol. 14, No. 1, pp. 131-155, First Quarter, 2012.

[13] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, “Scalable High Speed IP Routing Lookups,” *Proc. ACM SIGCOMM*, pp. 25-35, 1997.

[14] J. Mun, H. Lim and C. Yim, “Binary Search on Prefix Lengths for IP Address Lookup,” *IEEE Communications Letters*, Vol. 10, No. 6, pp. 492-494, June 2006.

[15] J. Mun, and H. Lim, “New Approach for Efficient IP Address Lookup Using a Bloom Filter in Trie-Based Algorithms," *IEEE Trans. on Computers*, Vol. 65, No. 5, pp. 1558-1565, May 2016.

저 자 소 개

변 하 영 (Hayoung Byun)
2014년 2월 이화여자대학교 전자공학과 졸업(학사). 2014년 3월~현재 이화여자대학교 전자전기공학과(석박사통합과정). 관심분야는 라우터나 스위치 등의 네트워크 관련 알고리즘 및 구조 설계, 콘텐츠 중심 네트워크(CCN).

임 혜 숙 (Hyesook Lim)
1986년 서울대학교 제어계측공학과 졸업(학사). 1986년 8월~1989년 2월 삼성휴렛 팩커드 연구원. 1991년 서울대학교 제어계측공학과 졸업(석사). 1996년 The University of Texas at Austin, Electrical and Computer Engineering 졸업(박사). 1996년 11월~2000년 7월 Lucent Technologies-Bell Labs, Member of Technical Staff. 2000년 7월~2002년 2월 Cisco Systems, Hardware Engineer. 2002년 3월~이화여자대학교 공과대학 전자전기공학과 정교수. 관심분야는 라우터나 스위치 등의 네트워크장비 설계 관련 알고리즘 및 하드웨어 구조 설계, 콘텐츠 중심 네트워크(CCN), 소프트웨어 정의 네트워크(SDN).